03063

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

21

POSGRADO EN CIENCIA E INGENIERIA DE LA COMPUTACION.

SIMULACION DE AMBIENTES ABIERTOS BASADOS EN MODELOS ATMOSFERICOS.

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS

P R E S E N T A :

MIGUEL MIRANDA MIRANDA

DIRECTORA DE TESIS. MAT. ANA LUISA SOLIS GONZALEZ COSIO

MEXICO, D.F. 2001

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

A mis padres
por su amor y apoyo incondicional.

A Guadalupe y Beatriz, mis sobrinas
porque desde su llegada he aprendido a ver el mundo en forma distinta.

Agradecimientos.

El realizar estudios de maestría fue una aspiración profesional y personal que determinó las experiencias, encuentros, amistades, pasatiempos y en fin, toda mi existencia en los últimos tres años. Deseo expresar mi agradecimiento a todas las personas que formaron parte de esta apasionante etapa de mi vida:

A Sergio, el "Archie", por ser el amigo que ha estado en los momentos más relevantes de mi vida, que goza con mis aciertos y sufre con mis tropiezos y además los acompaña con canciones a ritmo de los acordes de una guitarra. Gracias por la canción hermano!!!.

A Carlos Alberto, por hacer posible una de las experiencias más enriquecedoras que he vivido, conocer una cultura diferente. El recuerdo de las caminatas por St. Catherine, los paseos en bicicleta, la charla siempre inteligente, profunda y llena de reflexión ya sea en un café en Montreal o en una jardinera en Ciencias, son recuerdos muy especiales que han provocado cambios trascendentales en la forma en que percibo al mundo. Gracias hermano por tu apoyo constante ya sea en la cercanía o en la distancia.

A Antonio, mi hermano menor, por apoyarme desde el principio en esta aventura y compartiendo siempre el orgullo de ser universitario.

A Rocío y Ricardo mis amigos y compañeros de IMTSA, por alentarme y apoyarme para dar el paso inicial en la maestría.

A Moisés y Olivia quienes me brindaron su amistad y con quienes disfruto mucho conversando. De ellos he aprendido a explorar nuevos horizontes en la música y en la literatura.

A mis amigos y compañeros ingenieros de la ENEP Aragón, por su amistad, apoyo e interés en la culminación de mis metas. Gracias Raquel, Julius, Moni, Oved, Lucy, Malena, Alex, Matus, Charly, Arturo, Gato, Margara, Hugo, Lucas, Andrés...

A mis amigos y compañeros de la maestría por todos los momentos de alegría que compartimos en salones de clase, laboratorios, fiestas, reuniones, excursiones: Gracias Lolita, Sandra, Blanca, Carmen, Maria Elena, Mari, Judith, Alicia, Jaquie, Martín, Pablo, Juan Carlos, Jorge, Gabriel, Elio, Pepe...

Quiero expresar un sincero agradecimiento a mi asesora de tesis Ana Luisa Solís por haber abierto el camino que me llevo a descubrir mi gusto por el trabajo en el área de Graficación por Computadora. Desde que la conocí ha compartido su pasión por la investigación en esta área, enseñándome una nueva actitud hacia el trabajo y hacia la vida en general. Gracias por las conversaciones, las experiencias y apoyos que me ha brindado a lo largo de todo este tiempo.

Un sincero agradecimiento para los miembros del jurado que revisaron el presente trabajo enriqueciéndolo con sus puntos de vista:

Dra. Hanna Oktaba.
Dr. Boris Escalante Ramírez.
Dr. Homero V. Ríos Figueroa.
Dra. Genevieve Lucet Lagrifoul.

Un agradecimiento para todos los profesores que compartieron sus experiencias con nosotros.

Indice.

Introducción.

Una de las principales etapas en la generación de imágenes por computadora, es la de simular la interacción de la luz que incide sobre la superficie de los objetos en una escena. Los primeros algoritmos que se desarrollaron fueron el de H. Gouraud en 1971 y el de Phong Bui-Tuong en 1975. Los modelos de Gouraud y de Phong son conocidos como *modelos de iluminación local* porque consideran únicamente, la iluminación directa proveniente desde las fuentes de luz hacia la superficie de los objetos.

Posteriormente, se propusieron otros métodos en los que además de las fuentes de luz, se consideran las contribuciones provenientes desde todas las superficies del ambiente. En estos modelos todas las superficies proporcionan iluminación a la escena, al reflejar la luz proveniente tanto de las fuentes de luz, como de otras superficies. A estos modelos se les conoce como *modelos de iluminación global.*

Con esta diversidad de métodos es posible crear escenas de ambientes cerrados, como las habitaciones en el interior de una casa o edificio. Estos ambientes tienen la característica de que las fuentes de luz tienen posiciones bien definidas y sus contribuciones en la iluminación están delimitadas por el espacio en que están contenidas. Es por ello que la simulación de iluminación que realizan los modelos locales y globales, es suficiente para generar imágenes que simulan muchos de los efectos de la luz que interactúa con objetos del mundo real.

Con estos métodos se comenzaron a crear también, ambientes abiertos en los cuales la principal fuente de iluminación es el sol. Sin embargo estos ambientes requieren de propiedades adicionales a la iluminación; una de ellas es la simulación de la atmósfera terrestre. En toda escena de un espacio abierto, debe aparecer el color del cielo como fondo. Para darle una solución a este problema, inicialmente se coloreó el fondo con un azul uniforme, sin embargo las escenas así generadas, lucen poco naturales o realistas. Observando el cielo, podemos darnos cuenta que en la mayor parte del día, el cenit aparece con un azul brillante y el horizonte tiende hacia el color blanco. Una idea propuesta para conseguir este efecto, es colorear el cenit del ambiente en azul y el horizonte en blanco, posteriormente se realiza una interpolación para determinar el color del espacio entre ellos. En los últimos años se han propuesto otras soluciones a este problema, las cuales consideran los efectos que sufre la luz solar al atravesar la atmósfera terrestre.

El *principal objetivo* de este trabajo es presentar una propuesta para la simulación de ambientes abiertos, que incluya un modelo que logre el efecto de la coloración del cielo a diferentes horas del día. Un segundo objetivo planteado, fue el desarrollar un prototipo capaz de generar animaciones de un ambiente virtual, que permita comprobar la utilidad del modelo propuesto.

Gran parte del material presentado en el trabajo, está relacionado con la plataforma de software utilizada en el desarrollo del prototipo: la *interfaz RenderMan*. Esta interfaz fue diseñada específicamente para describir ambientes virtuales que serán generados por una computadora. La diferencia más notable de *RenderMan*, comparada con otras plataformas, es la definición de un lenguaje capaz de modelar el comportamiento de las superficies, fuentes de luz y fenómenos atmosféricos. Este lenguaje se conoce como lenguaje de "shading". Existen dos programas de "rendering", capaces de generar imágenes de alta calidad, interpretando descripciones realizadas con la *interfaz RenderMan*: *PhotoRealistic RenderMan y Blue Moon Rendering Tools*. Las imágenes del prototipo fueron creadas utilizando el segundo de ellos debido a que es software de distribución libre con la facilidad adicional de obtenerse para varias plataformas de hardware.

El trabajo se divide en 4 capítulos:

En el capítulo 1, se define lo que es la *interfaz RenderMan*, cuales son sus componentes y se mencionan los algoritmos con los que fueron diseñados, los dos programas de "rendering", basados en *RenderMan*, más utilizados: *PhotoRealistic RenderMan y Blue Moon Rendering Tools*.

En el capítulo 2, se describe el lenguaje de "shading" de *RenderMan*, herramienta con la que se modela el comportamiento de las superficies, volúmenes y fuentes de luz.

En el capítulo 3, se describe el modelo atmosférico que se utiliza para simular la coloración al cielo. El modelo considera las variaciones de densidad atmosférica provocadas por la altitud y emplea una simplificación del fenómeno conocido como dispersión Rayleigh. Un cielo sin nubes parecería poco natural, por ello se incluye un algoritmo para generación de nubes.

En el capítulo 4, se expone como se programaron, en lenguaje de shading, los modelos presentados en el capítulo 3. Para comprobar su utilidad práctica, estos modelos se aplicaron en la simulación de un ambiente abierto. Con este ambiente se aprecia una posible área de aplicación. En este capítulo se describe también, la realización de una animación del ambiente virtual, su programación en C++, las dificultades encontradas en su desarrollo, junto con las propuestas de solución.
Finalmente se hacen algunas conclusiones basadas en los resultados de la aplicación del modelo.

Capítulo 1.

Describiendo modelos con RenderMan.

En este capítulo se explica que es la *interfaz RenderMan*, su filosofía, cuáles son sus componentes y la forma en que organiza las actividades en la generación de imágenes por computadora. Se mencionan los algoritmos utilizados en los dos programas de "rendering" más famosos que cumplen con la especificación de la *interfaz RenderMan*: *PhotoRealistic RenderMan y Blue Moon Rendering Tools*. Los temas de este capítulo se basa en las referencias [2], [24], [26] y [28].

1.1 La filosofía de RenderMan.

RenderMan fue creado a finales de los años ochenta, en lo que más tarde se convertiría en la compañía *Pixar*, lugar donde se han creado las películas de dibujos animados por computadora más populares, como son "Toy Story", "A bugs life", "Luxo Jr", "Geri´s Game", entre otras. En aquellos años ya se disponía de diversas herramientas de software para la generación de imágenes, sin embargo eran muy limitadas y en muchos casos, no ofrecían facilidades para integrar nuevos algoritmos que permitieran expandir sus capacidades. Además era muy problemático utilizar varias de ellas en forma conjunta, porque no existía un mecanismo para intercambiar información de manera uniforme.

La idea que inspiró la creación de *RenderMan* surgió con la aparición de *Postscript*, el lenguaje de descripción de páginas. *Postscript* hizo posible la independencia de dispositivos de impresión. Ahora un documento puede capturarse en cualquier herramienta de software: editores de texto o programas de composición (TeX, LaTeX) e imprimirse con las mismas características y de igual calidad, en impresoras de diferentes fabricantes; siempre y cuando tanto las herramientas de captura como las impresoras, cumplan la especificación definida en el estándar *Postscript*. Así, *Postscript* cumple la función de intermediario entre programas de captura de texto e impresoras. *RenderMan* utiliza este mismo concepto [3].

RenderMan es la interfaz que comunica a los programas de modelado con los programas de "rendering". Los primeros son conocidos como modeladores. Su labor es facilitar el diseño, creación y modificación de figuras geométricas, así como la asignación de la posición que ocuparán dentro de un ambiente virtual. Estos programas deben tener interfaces gráficas muy amigables. Deben permitir la interacción constante con el usuario para que este pueda

manipular las figuras geométricas con facilidad y naturalidad. Para ello deben hacer uso de todo tipo de controles como ventanas, menús, barras de herramientas.

Los programas de "rendering" deben generar imágenes o vistas del ambiente virtual. Toman como referencia un punto en el ambiente, en donde es colocado un observador virtual; y calculan los atributos de los elementos participantes, como la iluminación, sombras, texturas, superficies visibles de los objetos, etc. con respecto a ese punto. El resultado es una imagen de la escena desde el punto de vista del observador. Haciendo una analogía con el mundo real el proceso de "rendering" es similar a la toma de fotografías. Un fotógrafo selecciona una cámara, la película, la iluminación, la posición desde donde se harán las tomas y al oprimir el obturador obtiene una imagen plasmada en un cuadro de la película.

El proceso de "rendering" se subdivide en varias etapas. Para cada una de ellas existen diversos algoritmos. Al diseñar y construir un programa de "rendering" la eficiencia de los algoritmos debe ser un factor fundamental. Debe permitirse el crecimiento del número y complejidad de los elementos participantes en un ambiente, sin incrementar demasiado el tiempo de cálculo de las imágenes. A diferencia de los modeladores, los programas de "rendering" no requieren de la interacción constante con el usuario por lo que no es necesario el uso de controles o ventanas para su utilización. Los dos principales programas de "rendering" que mencionaremos más adelante, no cuentan con una interfaz gráfica. La interacción se realiza a nivel de línea de comandos, como en MS-DOS o en un "shell" de UNIX.

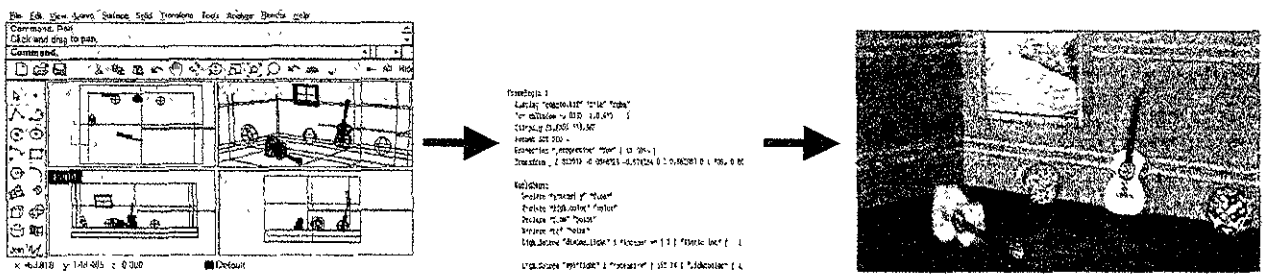


Figura 1.1. El papel de *RenderMan* como interfaz entre modeladores y programas de "rendering".

Con la separación de los procesos de modelado y de "rendering", *RenderMan* permite la independencia tanto de los dispositivos como del software utilizado en ambos procesos. Además, al dividir con claridad dos tareas que requieren metodologías completamente distintas, estimula a los programadores a construir software con una mayor especialización.

La descripción completa de *RenderMan* se publica en un documento conocido como la *Interfaz RenderMan* (*The RenderMan Interface*) [24] y [26], el cual se revisa y modifica constantemente. La versión 3.2 es la más actual. Este documento especifica qué información es necesaria para describir una escena tridimensional con alto grado de realismo. Dada la riqueza de elementos que se puede describir con *RenderMan* el documento es muy extenso. Sus creadores no exigen que un programa de "rendering" debe de construirse tratando de

cumplir todos los puntos de la especificación. En su lugar dividen el total de características que se pueden definir, en dos grupos:

- *Características requeridas:* que son las necesarias para generar imágenes básicas.
- *Características opcionales:* que sirven para incrementar el nivel de realismo en las imágenes.

Se dice que un programa cumple o satisface la *interfaz RenderMan* si incluye todas las características requeridas.

1.2 La interfaz RenderMan.

La interfaz describe las escenas por medio de un conjunto de procedimientos que alimentarán a un programa de "rendering". El programa debe ser capaz de interpretarlos y ejecutarlos. Los procedimientos se pueden declarar de dos maneras diferentes:

- *Utilizando una biblioteca en lenguaje C.* La especificación se diseñó originalmente como un conjunto de bibliotecas en C [28].
- *Utilizando un formato especial llamado RIB. (RenderMan Interface ByteStream Protocol).* Este formato cumple dos misiones: como formato de archivos y como protocolo de red para el transporte de secuencias de llamadas a una biblioteca *RenderMan*. Con esta última se puede dividir la tarea de generación de imágenes a través de una red de computadoras.

Existe una gran cantidad de procedimientos para definir una escena, cada uno de ellos con una tarea específica: describir la geometría, aplicar transformaciones geométricas, manipular la cámara, definir las texturas que se emplearán, las fuentes de luz etc. [2], [24], [26] y [28].

Además de contener todos los procedimientos para la descripción de una escena, el documento describe un lenguaje especial para la definición de texturas, diseño de fuentes de luz, deformaciones y otros atributos que enriquecen la generación de imágenes, conocido como lenguaje de "shading". Este lenguaje constituye una extensión de la interfaz.

1.3 El lenguaje de "shading".

"Shading" es el proceso de modelar la interacción de la iluminación, considerando la posición, orientación y propiedades de las superficies; con la finalidad de calcular el color y la intensidad de la luz en cada punto de la imagen que se desea generar. La iluminación se crea, a su vez, utilizando modelos de iluminación que pueden ser locales o globales. En los modelos locales, se considera únicamente la iluminación que proviene de las fuentes de luz. En los modelos globales, además de las fuentes de luz, se toma en cuenta la iluminación indirecta proveniente de la luz reflejada por todos los objetos del mundo virtual.

El modelo de iluminación local más ampliamente utilizado es el modelo de *iluminación de Phong*, que considera tres términos para su cálculo: *ambiental, difuso y especular*. Este modelo se expresa como:

$$C_{output} = K_a C_{amb} + \sum_{i=1}^{nlights} (K_d C_{diff}(N \bullet L_i) Cl_i + K_s C_{spec}(R \bullet L_i)^n$$

donde:

- L_i y Cl_i son la dirección y el color, respectivamente de la luz número i.
- K_a, K_d, K_s, n, C_{amb}, C_{diff}, y C_{spec} son parámetros especificados por el usuario. K_a y C_{amb} son el coeficiente de reflexión ambiental y el color de la iluminación ambiental respectivamente, K_d y C_{diff} son el coeficiente de reflexión difusa y el color de la iluminación difusa, por último K_s y C_{spec} son el coeficiente de reflexión especular y el color de la iluminación especular. Cambiando estos parámetros, se puede hacer que los objetos parezcan hechos de diferentes materiales.
- N es la normal de la superficie y R es la dirección de reflexión con respecto al punto de vista de la cámara.
- C_{output} es el color resultante de la superficie.

Muchos de los programas de modelado incluyen la opción de "rendering" que utilizan solamente el *modelo de Phong* combinado con el uso de texturas, y el "bump mapping" para el proceso de "shading". Las texturas son imágenes bi-dimensionales que se mapean en la superficie de los objetos, haciendo una analogía con el mundo real, este proceso es equivalente a cubrir los objetos con una envoltura de papel. "Bump mapping" es la creación de discontinuidades o rugosidades en la superficie, alterando los vectores normales de la superficie previo al cálculo del color de la superficie.

El nivel de realismo en las imágenes se puede mejorar en *RenderMan* gracias al uso del lenguaje de "shading", en donde se pueden definir modelos de iluminación, texturas y "bump mapping" más complejos. En *RenderMan* el usuario tiene la responsabilidad de definir el modelo de "shading" a emplear, con la ventaja de poderlo hacer tan rico y complejo como desee.

La idea original de un lenguaje de "shading" la propuso Cook[7] con los árboles de "shading". Estos árboles, son estructuras que ordenan el proceso que calcula los colores que deben de tener las superficies de los objetos, que forman parte de un ambiente virtual. Para determinar el color de la superficie de una figura geométrica, debemos tomar en cuenta diversas propiedades como la geometría, el tipo de material de que esta hecha, el medio ambiente, el color de la iluminación. Cada una de estas propiedades se conoce, en el modelo de Cook, con el nombre genérico de parámetro de apariencia.

Un árbol de "shading" contiene en sus nodos operaciones. Cada nodo puede recibir ninguno, uno o más parámetros de apariencia como entrada y puede producir uno o más parámetros como salida. El cálculo de "shading" se lleva a cabo recorriendo el árbol en postorden. La salida de la raíz del árbol dará el color y la opacidad final de la superficie. En [7] aparecen definidos tres diferentes tipos de árboles que, como veremos más adelante, son fundamentalmente los mismos tipos de "shaders" que existen en *RenderMan*:

- *Shade Trees*
- *Light Trees*
- *Atmosphere Trees*

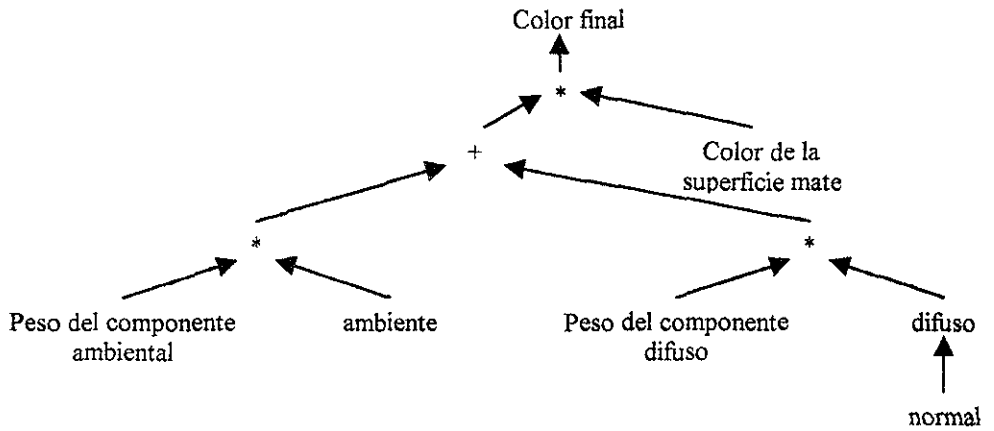


Figura 1.2. Árbol de "shading" para una superficie mate.

El lenguaje de "shading" es un lenguaje con sintaxis similar a C, con el que se definen y declaran funciones especiales conocidas como "shaders" que describen la luz emitida por las fuentes de luz y la atenuación que sufren por su interacción con las superficies y los volúmenes. Los volúmenes son espacios tridimensionales dentro de los cuales se modelan fenómenos como humo, niebla, neblina y en general todo tipo de perturbación atmosférica.

Existen varios tipos de "shaders" y estos se invocan desde la *interfaz RenderMan* utilizando una llamada de función especial específica para cada tipo:

- *"Shaders" de superficie:* describen la apariencia de las superficies y como interactúa la luz que incide en ellas. La invocación desde la interfaz es a través de la llamada a función RiSurface.

- *"Shaders" de desplazamiento:* Describe perturbaciones en las superficies como las rugosidades. La invocación desde la interfaz es a través de la llamada a función RiDisplacement.

- *"Shaders" de fuentes de luz:* Describe la dirección, intensidad y color de la luz que proporciona una fuente de luz. La invocación desde la interfaz es a través de la llamada a función RiLightSource.

- *"Shaders" de volumen:* Describen como es afectada la luz al pasar a través de un medio atmosférico como humo o niebla. La invocación desde la interfaz es a través de la llamada a función RiAtmosphere.

- *"Shaders" para imágenes finales:* Describen las transformaciones de color que sufren los pixeles obtenidos al final del proceso de "shading", pero antes de que se muestren en

pantalla o se escriban en un archivo. La invocación desde la interfaz es a través de la llamada a función RiImager.

Para poder utilizar un "shader" después de escribirlo, este se debe de compilar para generar archivos con "byte-codes" que serán la entrada a un interprete, que los ejecuta durante el proceso de "shading" de la imagen.

El modelo de ejecución de los "shaders" es de tipo implícito, es decir que al aplicarse un "shader" sobre una figura geométrica, los cálculos contenidos en el "shader" se realizarán en todos y cada uno de los puntos contenidos en dicha figura. Así por ejemplo, si queremos trazar una línea sobre un polígono, como el "shader" recorre todos los puntos sobre el polígono, se deben especificar las pruebas necesarias para determinar si el punto que actualmente se está analizando esta contenido o no en la línea. Utilizando un modelo explícito, bastaría con especificar en que puntos inicia y termina la línea deseada para dibujarse.

1.4 Programas de "rendering" que cumplen con la interfaz RenderMan.

Existen varios programas de "rendering" que cumplen con la interfaz, sin embargo dos sobresalen por su calidad:

- *PhotoRealistic RenderMan,* conocido por su abreviatura *PRMan* [25] el cual fue realizado por los estudios Pixar

- *Blue Moon Rendering Tools* conocido como *BMRT* programado por Larry Gritz [14].

Ambos programas incluyen diferentes características opcionales de la *interfaz RenderMan* por lo que a veces su compatibilidad no es total. Además estos programas reflejan claramente la independencia en cuanto a los algoritmos utilizados para la generación de imágenes. Mientras PRMan utiliza una arquitectura conocida como REYES [9], BMRT usa procedimientos de "Ray Tracing" y Radiosidad [15],

1.5 La arquitectura REYES.

REYES es la abreviatura de "*Renders Everything You Ever Saw*". Sus creadores tenían como propósito crear una arquitectura óptima para el "rendering" de imágenes complejas de alta calidad [9]. Originalmente fue diseñada y programada en hardware altamente especializado en el manejo de gráficos, pero posteriormente fue probada en equipos más modestos mostrando un buen desempeño. Por esta razón el grupo de desarrollo de *Pixar* decidió crear el primer programa de "rendering" que cumpliera con la *interfaz RenderMan* utilizando esta arquitectura.

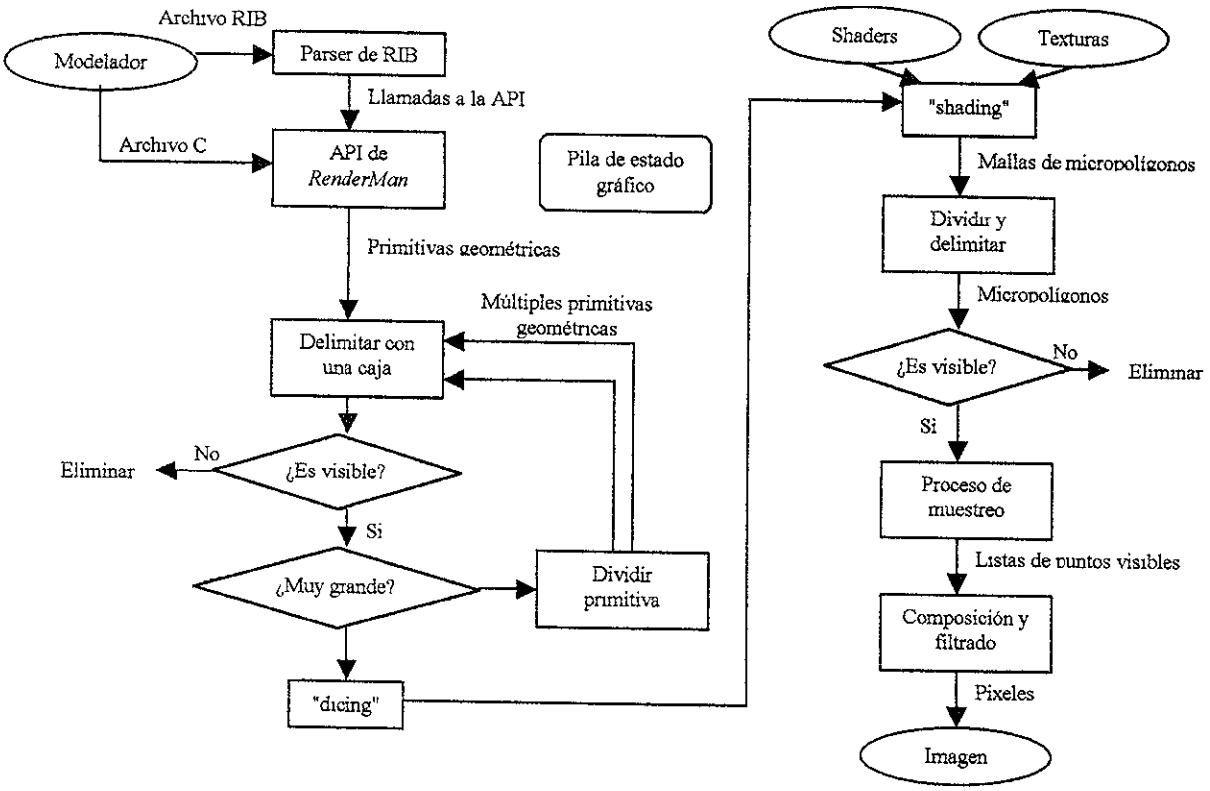


Fig. 1.3 Diagrama de la arquitectura *REYES*.

Ambos trabajos, la creación de la *interfaz RenderMan* y el primer programa de "rendering" *PhotoRealistic RenderMan*, fueron desarrollados en forma paralela.

1.5.1 Lectura del modelo.

El primer paso para construir una imagen es leer el modelo del mundo virtual. El modelo se genera en un modelador que produce un archivo RIB o un archivo en C.

1.5.2 Llamadas a la API.

Después se deben hacer las llamadas a la API del programa de "rendering". Si se proporciona un archivo RIB debe pasar por un "parser" que convierta cada uno de los procedimientos a su equivalente a una llamada API. Si es un archivo en C, la llamada a la API es de manera directa. En esta etapa se construye y mantiene la pila conocida como estado gráfico que contiene los atributos a utilizarse para dibujar los elementos geométricos, de la escena. Cuando se dibuja un elemento geométrico toma los atributos que se encuentran al tope del estado gráfico para dibujarse.

1.5.3 División de las primitivas.

Cada vez que *REYES* lee un elemento geométrico, que de ahora en adelante denominaremos como primitiva geométrica, se delimita con una caja cuyas aristas están alineadas con los ejes coordenados de la cámara. Está caja debe contener totalmente a la primitiva y se utiliza para hacer todos los cálculos posteriores de manera más sencilla. Después se verifica si la caja generada puede visualizarse completamente en pantalla. Esta prueba se puede realizar porque en *RenderMan* antes de definir cualquier primitiva geométrica se deben definir las características de la cámara: posición, campo de visión, profundidad de campo, proyección utilizada, entre otros. En base a estas características se calcula el volumen de visión. Todas las primitivas contenidas en este volumen serán visibles, el resto no serán tomadas en cuenta por la arquitectura. Si sólo una parte de la primitiva esta contenida en el volumen de visión, se hace una prueba de tamaño. Si es muy grande para visualizarse, se divide en primitivas más pequeñas, con la intención de crear subprimitivas que puedan contenerse dentro del volumen de visión. Cada subprimitiva generada se separa de la primitiva que la originó, para volver a pasar por la prueba de tamaño; si esta nueva subprimitiva es demasiado grande se vuelve a subdividir y así sucesivamente, hasta que en algún momento alguna de las subprimitivas generadas al pasar por la prueba de tamaño es lo suficientemente pequeña para visualizarse.

1.5.4 "Dicing".

Eventualmente todas las subprimitivas cumplirán con la prueba de tamaño y entonces pasarán a la siguiente fase conocida como "dicing". En esta etapa se transforman todas las primitivas a un formato común: la malla. Una malla, en esta arquitectura, es una cuadricula o arreglo rectangular de caras poligonales conocidas como micropolígonos.

El área de un micropolígono es igual o menor al área de un pixel, esto con la finalidad de hacer el mayor número de cálculos posibles para obtener imágenes con gran nivel de detalle. Posteriormente los cálculos de "shading" se harán sobre los vértices de los micropolígonos,

1.5.5 "Shading".

Cada una de las mallas pasa entonces al proceso de "shading". Para calcular el color de las caras de los micropolígonos, primero se hace el "shading" en todos sus vértices. Para ello se utilizan los "shaders".

El sistema de "shading" está programado como un interprete que recibe los "byte-codes" generados en la compilación de los "shaders". El interprete sigue una arquitectura de trabajo SIMD (Single Instruction, Multiple Data), esto quiere decir que, por cada línea de instrucción

que lee de los "byte-codes", ejecuta el operador en todos los vértices de la malla. Es mportante el orden en que se calculan los "shaders". En primer lugar se ejecutan los shaders" de desplazamiento, debido a que pueden deformar las primitivas geométricas o modificar las normales de los puntos en la superficies. Después se evalúan los "shaders" de superficie. En casi todos estos "shaders" se hace el cálculo del modelo de iluminación. Para calcularlo suele requerirse de llamadas a las funciones que obtienen los términos difuso y especular; estas funciones necesitan, a su vez, la evaluación de los "shaders" de fuentes de uz que intervienen en la superficie que se está evaluando. Por lo tanto los "shaders" de uentes de luz se ejecutan como co-rutinas de los "shaders" de superficie. Los "shaders" de uentes de luz son evaluados solamente la primera vez que se invocan. Su resultado se almacena para su uso posterior. Finalmente se calculan los "shaders" de volumen que pueden cambiar el color y la opacidad de los vértices. Es importante aclarar que los "shader" para imágenes finales no intervienen en el proceso de "shading" de esta arquitectura, ya que como se describió anteriormente se aplican en la etapa final del "shading", a nivel de pixeles, cambiando su color antes de dibujarse en pantalla o escribirse en un archivo.

1.5.6 Determinación de superficies visibles, combinación y muestreo.

Después del "shading", la malla pasa a la rutina de determinación de superficies visibles. El primer paso dentro de esta rutina es romper la malla en micropolígonos individuales. Entonces utilizando una versión similar del proceso de "dicing", delimitamos cada micropolígono con una caja, para checar si es visible. Si no lo es lo eliminamos del proceso.

Por otra parte debemos de realizar un proceso de muestreo estocástico conocido como "jittering" [8] en el espacio de pantalla que utilizaremos para construir la imagen final. Este espacio es la matriz que define la resolución en pixeles del monitor o del archivo que contendrá la imagen. Todo proceso de muestreo de puntos, irremediablemente genera efectos de "aliasing". Por ejemplo cuando se dibujan en un monitor líneas rectas diagonales, estas se construyen utilizando una sucesión de pixeles; que al mirarse con detalle se observa que no es una línea perfectamente delineada. En su lugar obtenemos una recta con picos que se percibe como una escalera. El proceso de "jittering" disminuye el problema convirtiendo el fenómeno de "aliasing" en otro fenómeno menos evidente para el ojo humano, la generación de ruido. Para llevar a cabo el muestreo primero debemos determinar los lugares, en cada uno de los pixeles, donde se tomarán las muestras. Para ello se dividen los pixeles en cuadriculas, por lo general de 4X4, donde cada cuadrado resultante es conocido como subpixel. Cada subpixel tiene un punto de muestreo que se localiza en su centro. El proceso de "jittering" consiste en adicionar un desplazamiento aleatorio a la localización del punto de muestreo, para tomar muestras a intervalos irregulares, evitando la generación de patrones indeseables en la imagen final [9].

El concepto de "jittering" en este contexto es completamente diferente al que se utiliza en dispositivos de captura de audio y vídeo. En estos casos cuando se habla de "jittering" se trata de un error que tiene su origen en la variación del tiempo de retardo de una señal que viaja a través de una diferentes circuitos electrónicos. Todos los dispositivos digitales que tienen una entrada y una salida pueden adicionar "jitter" en la trayectoria de la señal, o en

otras palabras la señal original sufre distorsiones por las diferentes impedancias existentes en la trayectoria de la señal.

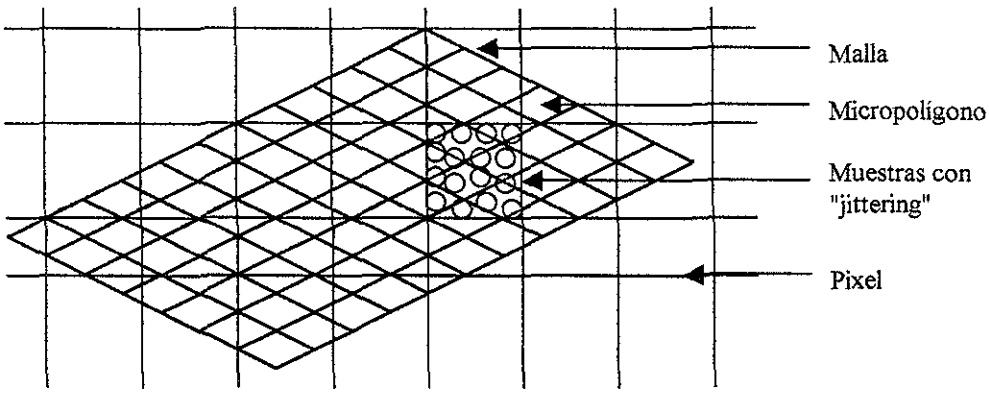


Fig 1.4 Sobreposición de la malla de micropolígonos en los pixeles.

Regresando a la rutina de determinación de superficies visibles; teniendo por un lado las localizaciones de los puntos de muestreo y por el otro los micropolígonos, estos últimos se transforman al espacio de pantalla para sobreponerse a los pixeles y así obtener su color y opacidad correspondiente. Para ello todas las muestras que están bajo un micropolígono toman el color y la opacidad de dicho micropolígono. Además registran la profundidad del micropolígono con respecto a la cámara. Estos tres valores se almacenan en una unidad conocida como punto visible. Dado que cada muestra puede estar cubierta por varios micropolígonos, que corresponden a diferentes primitivas, se debe asignar una lista de puntos visibles a cada muestra.

Recordemos que el proceso de "shading" se calculaba en los vértices de los micropolígonos. Para obtener el color de la superficie del micropolígono podemos usar dos métodos diferentes:

- Tomar los valores para el color y opacidad, obtenidos en el vértice localizado más a la izquierda del micropolígono.

- Calcular color y opacidad de la superficie por *interpolación de Gouraud*, que consiste en calcular primero colores y opacidades en las aristas del micropolígono por interpolación lineal y posteriormente calcular los valores en la superficie con la ayuda de líneas horizontales de rastreo sobre las cuales se aplica interpolación lineal tomando los valores de las aristas recientemente calculadas.

En un mundo virtual complejo pueden existir una cantidad enorme de objetos los cuales pueden sobreponerse, ocultándose unos con otros, por lo tanto para calcular el color y opacidad de un pixel se deben considerar todos los objetos que dada su posición se localicen en dicho pixel. Una vez que todas las primitivas que cubren un pixel se han procesado, se debe determinar el color de cada muestra en base a los puntos visibles contenidos en su lista, y finalmente las muestras resultantes se deben de combinar utilizando un filtro de reconstrucción para obtener los valores definitivos de color y opacidad de cada pixel.

```
nicializa el z buffer.
Para cada primitiva geométrica en el modelo,
    Leer la primitiva del archivo que contiene el modelo
    Si la primitiva puede delimitarse con una caja,
        Delimitar la primitiva en espacio de cámara.
        Si la primitiva esta completamente fuera del volumen de visión,
            Eliminarla.
        Si la primitiva puede dividirse,
            Marcar la primitiva como no lista para "dicing".
        Si no
            Convertir la caja delimitante a coordenas de pantalla.
            Si la caja está fuera de la pantalla,
                Eliminarla.
    Si la primitiva esta lista para "dicing",
        Convertir la primitiva en una malla de micropolígonos.
        Calcular los vectores tangentes y normales para los micropolígonos de la malla.
        Realizar el "shading" en los micropolígonos de la malla.
        Romper la malla en micropolígonos individuales.
        Para cada micropolígono,
            Delimitar el micropolígono con una caja en espacio de cámara.
            Si el micropolígono está fuera del volumen de visión,
                Eliminarlo.
            Convertir el micropolígono a espacio de pantalla.
            Delimitar el micropolígono en espacio de pantalla.
            Para cada punto de muestreo dentro del delimitador en espacio de
            pantalla,
                Si el punto de muestreo esta dentro del micropolígono,
                    Calcular la coordenada z del micropolígono en el punto de
                    muestreo por interpolación.
                    Si la coordenada z en el punto de muestreo es menor que
                    la coordenada z del buffer,
                        Reemplace la muestra en el buffer con esta muestra.
    Si no
        Dividir la primitiva en otras primitivas geométricas.
        Poner las nuevas primitivas en la primera posición de la porción no leída del
        archivo de modelo.
Filtre las muestras de los puntos visibles para producir los pixeles.
Salida de los pixeles.
```

Fig. 1.5 Pseudocódigo de la arquitectura *REYES*.

1.5.7 Problemas de la arquitectura REYES.

Una de las características que hace especial a esta arquitectura es que el proceso de "shading" se realiza antes de la determinación de superficies visibles. La ventaja de esta propuesta es que permite desplazar o deformar las superficies de las primitivas por medio de "shaders". Sin embargo esto tiene también una desventaja. Si se tiene una escena con muchos objetos sobrepuestos, dado que se realiza el "shading" para todas las primitivas, se realizarán cálculos en puntos que van a estar ocultos y que no tendrán participación en la imagen final, desperdiciándose mucho tiempo de cómputo.

También se debe tomar en cuenta el espacio en memoria que ocupa todo el proceso. Las primitivas se dividen, se fragmentan, y se muestrean. A cada paso de la arquitectura crece la cantidad de información que es almacenada. El último paso es el más demandante, recordemos que para determinar color y opacidad de un pixel se tienen que almacenar las listas de puntos visibles para todas las muestras correspondientes al pixel y que involucran a todas las primitivas que están en el espacio del pixel. Estas listas no se liberan de la memoria hasta que todas las primitivas se hayan procesado. Esto ocurre porque en ningún momento almacenamos la posición de las primitivas con respecto a los pixeles.

1.5.7.1 "Bucketing".

Para mejorar la arquitectura, se diseño una modificación que permite evitar el almacenamiento de gran cantidad de información en la memoria. Esta consiste en dividir la imagen en regiones rectangulares, conocidas como "buckets", las cuales serán procesadas completamente por separado. Con esta modificación las primitivas se ordenan en base al "bucket" en donde se localizan. Cada "bucket" tendrá una lista de primitivas. Si una primitiva no está en el "bucket" que se está procesando, el cual llamaremos "bucket" actual; está se guarda en la lista del primer "bucket" que la contiene; por lo tanto esta primitiva no pasará a la siguiente etapa, hasta que se procese el "bucket" en el cual fue reubicada, conservando su representación geométrica original sin expandirse en memoria. El algoritmo procesa un "bucket" a la vez. Los objetos que pertenecen al "bucket" actual pasan por el proceso de división o de "dicing". Las primitivas divididas se pueden adicionar a la lista del "bucket" actual o adicionarse a las listas de futuros "buckets", dependiendo de la caja que los delimita. Las mallas de micropolígonos, producto del proceso de "dicing", pasan por el proceso de "shading" para después ser separados. Durante la separación, los micropolígonos son delimitados por una caja y de manera similar, si no están en el "bucket" actual no pasan al proceso de muestreo hasta que se procese el "bucket" al que pertenecen. Eventualmente no habrá más primitivas en la lista del "bucket" actual. Por lo tanto todas las listas de puntos visibles en ese "bucket" se pueden procesar y liberarse de la memoria. Finalmente el "bucket" se puede mostrar en pantalla o almacenarse en archivo.

.5.7.2 Eliminación por ocultamiento.

)tro problema al que se le propuso una modificación, para mejorar el desempeño de la rquitectura, fue la determinación de superficies visibles. Ya se comentó que todas las rimitivas pasan primero por la etapa de "shading" y posteriormente a la determinación de uperficies visibles. La modificación consiste en ordenar las primitivas por profundidad con especto a la cámara en una lista, de tal manera que las primitivas más cercanas a la cámara e encuentren al principio y las más lejanas al final de dicha lista. Adicionalmente se debe ener una estructura de datos en memoria que registre el área del "bucket" que ha sido ubierta por los objetos opacos y su profundidad. Cuando el "bucket" esta completamente ubierto por objetos opacos, cualquier primitiva que este detrás de esos objetos no debe omarse en cuenta y no pasan a las etapas sucesivas del algoritmo, evitandose de esta orma cálculos innecesarios.

1.6 El empleo de los algoritmos de "Ray Tracing" / Radiosidad en BMRT.

BMRT es un programa de "rendering" que satisface la *interfaz RenderMan*. Se construyó con una arquitectura completamente diferente a *PRMan* [15]. La idea que impulso a Gritz el diseñarlo, fue demostrar que era posible crear un programa de "rendering" basado en *RenderMan* con algoritmos de iluminación global. El algoritmo que utiliza para generar imágenes es conocido como "Distributed Ray Tracing" que es la expansión de un algoritmo ampliamente utilizado en graficación: "Ray Tracing".

La primera persona que propuso la técnica de "ray tracing" para resolver el problema de 'shading" fue Whitted [31], que integró el cálculo de sombras, reflexiones y refracciones en su modelo. La idea esencial es simple: se trazan rayos primarios desde el punto donde se ocaliza el observador hacia el mundo virtual. El número de rayos se determina de acuerdo al número de pixeles que tendrá la imagen final. Desde los puntos donde ocurre una ntersección entre un rayo y una superficie, se traza un rayo secundario dirigido hacia una uente de luz. Si se encuentra un objeto entre la superficie y la fuente de luz, se determina que el punto de intersección esta bajo sombra y la intensidad del pixel que mostrará dicho punto disminuirá de acuerdo al modelo de iluminación.

También se pueden trazar nuevos rayos secundarios si la superficie es de material reflejante o transparente. En el primer caso si el rayo secundario reflejado intersecta a otra superficie, esta última se utiliza, en combinación con la primera superficie, para determinar el color del pixel de la imagen. Cuando se trata de un material transparente se traza un rayo secundario con un ángulo ligeramente modificado de acuerdo con el nivel de refractancia de la superficie, y se maneja de la misma forma que en el caso anterior: si intersecta otra superficie, se utiliza la combinación de ambas para determinar el color del pixel en la imagen.

La posibilidad de trazar nuevos rayos en superficies reflejantes y transparentes le adiciona recursividad al algoritmo. Para manejar en forma ordenada los rayos generados para cada pixel, se utiliza un árbol conocido como árbol de rayos, en donde las aristas indican la secuencia en que se lleva a cabo el proceso de "ray tracing" y los nodos contienen información acerca de como se realiza el proceso de "shading". Los cálculos que determinan el color que tendrá el pixel en la imagen se realizan al recorrer el árbol en forma ascendente. Al visitar cada nodo se realizan las operaciones de acuerdo a la información de "shading" que estos contienen y a la información que se obtuvo de cada nodo hijo. La altura del árbol la establece el usuario al especificar el nivel de rayos que pueden trazarse para calcular un pixel.

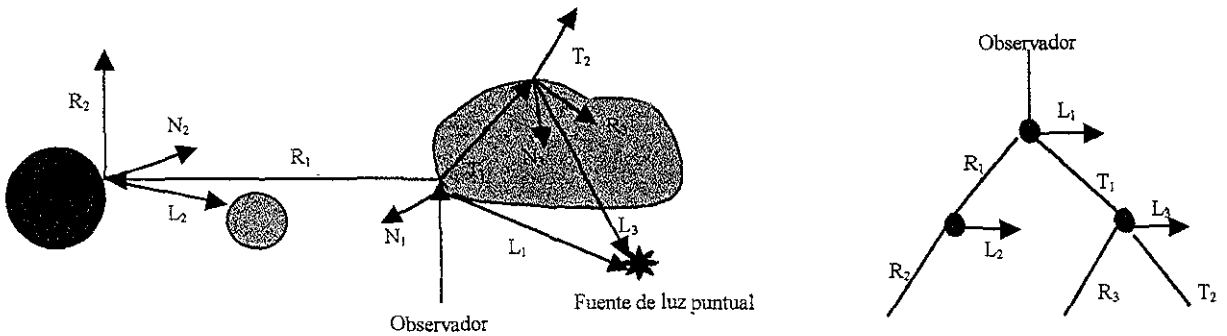


Fig. 1.6. Ejemplo del algoritmo de "Ray Tracing". A la derecha se muestra el árbol de rayos correspondiente. En ambas figuras N_i representa las normales a las superficies, R_i los rayos reflejados, L_i los rayos de sombra y T_i los rayos transmitidos.

El algoritmo "distributed ray tracing" fue propuesto por Cook [10]. El algoritmo de Whitted considera el trazado de rayos en un espacio tridimensional tomando en consideración el número de pixeles en la imagen. El algoritmo de Cook expandió esta idea distribuyendo rayos en otros espacios para conseguir diversos efectos.
Trazando rayos en el tiempo se consigue el efecto conocido como "Motion Blur"; este efecto aparece en fotografías de objetos en movimiento, los cuales aparecen borrosos. Trazando rayos con respecto a la posición del observador se consigue modificar la profundidad de campo y trazando rayos con respecto a las fuentes de luz se consigue el suavizado de sombras.

El algoritmo que utiliza *BMRT* para generar imágenes es el siguiente:

1. Al igual que en la arquitectura *REYES* el primer paso consiste en leer el archivo RIB que describe la escena. El archivo pasa por un parser que convierte cada procedimiento en su equivalente a una llamada de la API.

2. Se construye en forma automática, una jerarquía de cajas que delimitan la geometría. Estas cajas aceleran la traza de los rayos.

3. El empleo del método de radiosidad es opcional. Si este se realiza, la geometría se transforma en mallas. Entonces se emplea un algoritmo de radiosidad de refinamiento progresivo [29] [6].

Se realiza el trazado de rayos en la escena. Cuando se encuentra la intersección más cercana sobre el rayo con respecto al observador, se aplica el "shader" de desplazamiento y el "shader" de superficie, lo que da como resultado el color y opacidad de la superficie del objeto visible.

Capítulo 2.

El lenguaje de "shading".

En el capítulo anterior se mencionó que *RenderMan* define un lenguaje de "shading". Es importante conocer este lenguaje, porque nos brinda la posibilidad de modelar las características que determinan la apariencia de los elementos contenidos en un ambiente virtual, como son las superficies de los objetos, las fuentes de luz, las atmósferas, entre muchos otros. En el lenguaje de "shading" cada una de estas características se describe declarando funciones especiales, conocidas como "shaders". Existen varios tipos de ellos, sin embargo con tres de ellos se describe la apariencia general del ambiente: "shaders" de superficie, "shaders" de fuentes de luz, y "shaders" de volumen. En este capítulo revisaremos la sintaxis básica para la programación de "shaders". El material de este capítulo se basa en las referencias [2], [4], [24], [26] y [28].

2.1 Sintaxis del lenguaje de "shading".

La sintaxis del lenguaje de "shading" tiene muchas similitudes con la del lenguaje de programación C:

- La unidad de estructuración es la función.
- Se tienen las mismas sentencias de control.
- Tiene las mismas operaciones aritméticas y lógicas.

Sin embargo también tiene muchas diferencias:

- No existen las estructuras, uniones, apuntadores, arreglos. La falta de estos elementos limita al lenguaje ya que no se pueden crear estructuras de datos complejas.
- Tiene un conjunto tipos de datos predefinidos diferente al de C.
- No permite la creación de nuevos tipos de datos.

Una descripción completa del lenguaje de "shading" se puede consultar en [24] y [26]. A continuación, se mencionan las diferencias más notables entre el lenguaje de "shading" con respecto al lenguaje C.

2.1.1 Características de las funciones.

Las funciones en el lenguaje de "shading", tienen las siguientes características:

- Si no se declara ningún tipo en la definición de la función, el valor de retorno será float.
- Solamente se permite una sentencia return por cada función. La excepción a esta regla son las funciones void que no devuelven ningún valor y no requieren de la sentencia return.
- Todos los parámetros de la función se pasan por referencia.
- Esta prohibido el uso de la recursión tanto directa como indirecta: ninguna función puede llamarse a si misma, ni tampoco una función que fue invocada puede invocar a la función desde donde se invocó.
- No se pueden compilar funciones en forma independiente al cuerpo del shader. Las funciones se deben de declarar antes de usarse y se deben de compilar en la misma secuencia de compilación que el shader.
- Si esta permitido el uso del mecanismo #include para definir funciones en una archivo separado.

2.1.2 Características de los shaders.

Las funciones y los shaders son sintácticamente idénticos. Los shaders son funciones especiales que se ejecutan exclusivamente dentro de *RenderMan* y tienen las siguientes restricciones:

- Los shaders no pueden regresar valores, por tanto no pueden emplear la sentencia return.
- El acceso a los shaders esta restringido. Solo pueden ser invocados por el programa de "rendering", a través de la *interfaz RenderMan.*
- No pueden invocarse desde otros shaders ni tampoco desde funciones.

Existe una palabra reservada para especificar el tipo de shader que se esta definiendo:

- Shaders de superficie: surface
- Shaders de fuentes de luz: light
- Shaders de volumen: volume
- Shaders de desplazamiento: displacement
- Shaders para imágenes finales: imager

Esta palabra reservada debe preceder al nombre del shader y a la lista de parámetros. Al compilar el shader, el archivo resultante que contendrá los byte codes, tomará el mismo nombre que el shader, independientemente del nombre del archivo que contiene el código en lenguaje de "shading".

2.1.3 Tipos de datos.

float: al igual que en C.

string: Se utiliza para representar cadenas de caracteres que le dan nombre a los objetos externos. Solamente pueden ser constantes o variables de instancia.

point: Se utiliza para representar posiciones o direcciones en un espacio tridimensional. Un point es un vector de tres valores de tipo float. Por ejemplo: point p(2.5,2.5,2.5).
Los componentes individuales de un punto se accesan por medio de funciones especiales: xcomp(), ycomp(), zcomp().

color: Se utiliza para representar colores e intensidades de luz o la reflectividad y opacidad de una superficie. Es un tipo de datos abstracto que no tiene una estructura definida. Tampoco se aplica sobre un modelo de color en particular. Los modelos de color se representan por medio de sistemas coordenados. En el lenguaje de "shading" existen 5 modelos predefinidos:

- *"rgb":* rojo (red), verde (green), azul (blue).
- *"hsv":* tinte (hue), saturación (saturation), valor (value).
- *"xyz":* CIE coordenadas xyz
- *"xyY":* CIE coordenadas xyY.
- *"YIQ":* coordenadas NTSC.

Los valores de 0 en los componentes de color corresponden a la mínima intensidad, mientras que los valores de 1 corresponden a la máxima intensidad.
El sistema de coordenadas por omisión es el "rgb".
Un ejemplo de la definición de una variable de color en rgb es :
color e = color(0.5, 0.3, 0.2)

matrix: Se utiliza para representar transformaciones de matrices requeridas en la transformación de puntos y vectores de un sistema de coordenadas a otro. Las matrices se representan como un conjunto de 16 valores tipo float, que conforman una matriz de 4X4.

2.1.4 Modificadores de almacenamiento.

De acuerdo a la forma en que se almacen los valores en memoria las variables se clasifican en dos tipos:

- *uniform:* variables donde los valores son constantes sobre toda la superficie. En este caso se almacena un solo valor para toda la superficie y este valor puede ser modificado a lo largo del proceso de "rendering".

- *varying:* variables donde los valores cambian a lo largo de toda la superficie. Aquí se requiere almacenar un valor para cada uno de los puntos considerados en la superficie.

Las variables declaradas en una lista de argumentos de un shader por omisión asumen el modificador uniform. Las variables declaradas dentro del cuerpo del shader toman por omisión el modificador varying.

2.1.5 Variables globales.

Son variables predefinidas en el lenguaje de "shading". Contienen información, requerida por el "shader", acerca del punto que se esté procesando. Los valores que contienen gran parte de estas variables, son actualizados por el programa de "rendering" en forma automática. Para utilizar estas variables no se requiere su definición.
Existe un conjunto de variables globales para cada uno de los tipos de "shaders". Las variables para los "shaders" de superficie, de fuentes de luz y de volumen se listan en el anexo 1.

Las variables globales para los "shaders" de transformación y para imágenes finales se pueden consultar en [24], [26] y [28].

2.1.6 Operaciones sobre puntos.

Cuando se aplican los operadores aritméticos a dos operadores de tipo point o color, la operación se aplica a cada una de las parejas de los componentes de los operadores, para formar como resultado un valor del mismo tipo. No se permite realizar operaciones entre operadores de tipo point con operadores de tipo color a pesar de que suelen tener el mismo número de componentes.
Cuando se realiza una operación aritmética entre un operador de tipo float con uno de tipo point o color, se realiza el cálculo con cada uno de los componentes y el tipo float, como si aplicáramos un operador de tipo point o color cuyos componentes tuvieran valores idénticos.

Producto punto.
En el lenguaje de "shading" no existen las estructuras, por lo tanto el operador "." se le da la funcionalidad de producto punto. Toma dos operadores de tipo point y genera como resultado un valor de tipo float. El producto punto se calcula multiplicando cada pareja de componentes y sumando los resultados de los productos. Se utiliza principalmente para medir el ángulo entre dos vectores.

Producto cruz.
El operador para el producto cruz es "^". Se aplica a dos operadores de tipo point y da como resultado un valor también de tipo point. Si A y B son dos vectores representados

con variables de tipo point, el resultado de A^B será un vector perpendicular a ambos vectores A y B. El producto cruz se utiliza para encontrar la normal a una superficie en un punto dado, utilizando dos vectores *no paralelos, tangentes a la superficie del punto.*

2.1.7 Sentencias de control illuminance, illuminate y solar.

El Lenguaje de "shading" contiene, además de las conocidas sentencias de control empleadas en C, tres bloques de control adicionales: illuminance, illuminate y solar.

Illuminance esta disponible en los shaders de superficie. Permite integrar las propiedades de todas las luces que inciden en el punto sobre la superficie, en donde se este realizando el proceso de "shading".
Las variables globales: Cl color de la luz y L dirección de la luz, están disponibles exclusivamente dentro de un bloque illuminance.
Las dos formas de la sentencia illuminance son:

illuminance(position [, nsamples]) stmt
illuminance(position, axis, angle [, nsamples]) stmt

La primera forma especifica que la integral se extiende sobre una esfera que tiene como centro la posición definida por position. La segunda forma integra sobre un cono, cuyo ápice se localiza en la superficie, en la posición definida por position. Para construir el cono se utiliza la línea central definida por axis y el ángulo entre el borde del cono y la línea central, especificado en radianes, definido por angle.

Las sentencias illuminate y solar están disponibles en los shaders de fuentes de luz. Son inversas a la sentencia illuminance. Controlan la emisión de luz en diferentes direcciones. Dentro del bloque se puede disponer de la variable L que corresponde a la dirección de la luz. La variable Cl debe proporcionarla el programador y corresponde al color de la luz en la dirección definida por L.
La sentencia illuminate se usa para especificar la luz emitida por fuentes de luz locales.
Las formas generales son:

illuminate(position) stmt
illuminate(position, axis, angle) stmt

La primera forma especifica que la luz se emite en todas las direcciones. La segunda forma especifica que la luz sólo se emite dentro de un cono. La magnitud de L, dentro de la sentencia illuminate, es igual a la distancia entre la fuente de luz y la superficie en que se esta realizando el proceso de "shading".
La sentencia solar se usa para especificar la luz emitida por una fuente de luz distante.
Las formas generales son:

solar() stmt
solar(axis, angle) stmt

La primera forma especifica que la luz se emite desde todos los puntos en el infinito. La segunda forma especifica que la luz se emite desde todas las direcciones dentro de un cono, debido a que este cono especifica solo direcciones, no se requiere un ápice.

Estas tres sentencias de control no pueden anidarse.

2.1.8 Funciones integradas al lenguaje de "shading".

El lenguaje de "shading" contiene un conjunto de funciones matemáticas y geométricas predefinidas. Con ellas podemos expresar las operaciones que deben realizar los "shaders". Se puede consultar un breve listado de ellas en el anexo 2.

2.2 Sistemas de coordenadas predefinidos en el lenguaje de "shading".

En la graficación por computadora es natural tener varios sistemas de coordenadas de referencia. Por ejemplo, en un ambiente virtual, a cada objeto se le asigna un sistema de coordenadas único, con la finalidad de definir las características propias de él. Sin embargo, al formar parte de un ambiente, deben tener la referencia con respecto a un sistema de coordenadas global, llamada de mundo. Este sistema no permite determinar la posición de cada objeto con respecto a todos los demás objetos. Al definirse la posición donde se encontrará el observador el ambiente, los elementos se refieren con respecto a la cámara, etc.
Al programar un "shader", es importante determinar en cual espacio vamos a manipular la geometría. Los espacios predefinidos en el lenguaje de "shading" son :

- *"current"* : Es el sistema de coordenadas por omisión. Todos los puntos inicialmente están expresados en este sistema y los cálculos de iluminación se llevan a cabo también aquí. La determinación de este sistema de coordenadas es dependiente del programa de "rendering". En BMRT este sistema es el mismo que world, mientras que en PRMan es igual a Camera.
- *"object"* : Es el sistema de coordenadas local de las primitivas gráficas.
- "shader" : Es el sistema de coordenadas activas en el momento que se invoca al shader.
- *"world"* : Es el sistema de coordenadas global, activo cuando se declara la sentencia WorldBegin.
- *"camera"* : Es el sistema de coordenadas cuyo origen esta en el centro del lente de la cámara virtual, el eje x positivo se dirige hacia la derecha, el y positivo hacia arriba y el z positivo hacia dentro de la pantalla.
- *"screen"* : Es el sistema de coordenadas del plano de proyección de la imagen. Solo tiene dos coordenadas por tratarse de un plano. La coordenada (0, 0) en este sistema esta dirigido hacia el eje z del espacio de camera.
- *"raster"* : Al igual que el anterior es el espacio de la imagen proyectada en 2 dimensiones, pero con la diferencia de que las unidades corresponden a los pixeles en pantalla. La coordenada (0, 0) en este sistema corresponde a la esquina superior izquierda de la pantalla. La coordenada x se incrementa hacia la derecha y la y hacia abajo.
- *"NDC"* : Coordenadas de dispositivo normalizadas. Es igual al sistema de coordenadas "screen" con la diferencia de que las coordenadas están normalizadas.

Los puntos que son procesados en un shader pueden ser fácilmente transformados de un sistema a otro con alguna de las variantes de la función transform.

2.3 Compiladores para el lenguaje de "shading".

Los programas de "rendering" incluyen un compilador para el lenguaje de "shading". En *PRMan* su nombre es shader y en *BMRT* es slc. En cada archivo que se desee compilar debe existir únicamente un "shader", el cual le dará el nombre al archivo de "byte codes", que se generará en la compilación. Los "shaders" puede hacer uso de cualquier numero de funciones, las cuales no necesariamente deben ser declaradas en el mismo archivo. El lenguaje de "shading" permite el mecanismo de inclusión de archivos utilizando la sentencia #include.
Todas las funciones programadas por el usuario, deberán ser declaradas en la misma secuencia de compilación del "shader" que las requiera, debido a que los compiladores no cuentan con un ligador. En otras palabras no se permite la creación de módulos independientes de compilación.
Los "byte codes" que se generan en compilación, son interpretados por el programa de "rendering" al momento de realizar el "shading".

2.4 Etapas en el proceso de "shading".

En el capítulo anterior mencionamos la existencia de 5 diferentes tipos de shaders.
Tres de ellos definen las etapas principales en el proceso de "shading":

- *"Shaders" de superficie:* Los "shaders" de superficie se aplican a las primitivas geométricas. Tienen como objetivo modelar las propiedades ópticas de los materiales con que se construyen las primitivas. Estos "shaders" calculan la luz reflejada hacia una dirección en particular, sumando la interacción de las luces que iluminan la superficie de la primitiva, considerando las propiedades del material con el que están hechas.
- *"Shaders" de fuentes de luz:* Un "shaders" de fuente de luz calcula el color de la luz que se emite, desde la posición en donde se encuentra la fuente hasta las superficies que esta alcanza a iluminar. Las propiedades que tiene la luz son: color o espectro, intensidad, dependencia direccional y disminución de intensidad con la distancia. Una fuente de luz puede relacionarse a una primitiva geométrica o definirse como una entidad que no posee geometría.
- *"Shaders" de volumen:* Modifican el color de un rayo de luz que viaja a través del interior de un objeto sólido. Los "shaders" de volumen más utilizados son los que calculan efectos atmosféricos, como niebla, nubes, neblina.

Un "shader" de fuente de luz tiene el objetivo de calcular la luz que toca un punto en la superficie de un objeto. Un "shader" de superficie usa el color de las luces provenientes de las fuentes de luz, y junto con las propiedades del material del objeto, calcula la luz

que refleja la superficie. Esta información alimenta a un "shader" atmosférico, el cual puede modificar la luz que llega al plano de proyección donde se formará la imagen final.

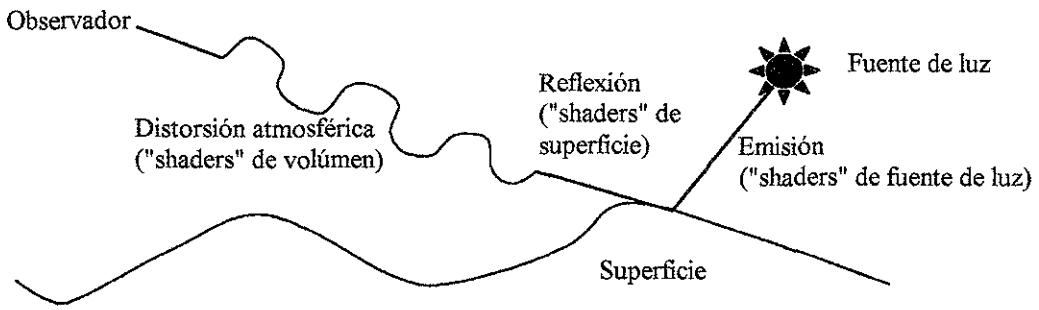


Fig. 2.1 Las tres partes básicas del proceso de "shading": emisión de la fuente de luz, interacción de la luz con la superficie, y efectos atmosféricos entre la superficie y el observador.

2.5 Mapeo de texturas y texturas procedurales.

En graficación por computadora, a los detalles que presenta la superficie de un objeto se le conoce como textura. El mapeo de texturas consiste en mapear una imagen bidimensional sobre una superficie tridimensional, con la finalidad de plasmar los detalles de la superficie. Esta fue la primera técnica que permitió agregar mayor realismo a las imágenes generadas por computadora. Ahora las superficies, además del color podían, tener patrones complejos grabados.

El disponer de un lenguaje de programación para especificar el comportamiento de las fuentes de luz y las superficies, hace posible su utilización en la determinación de la apariencia de los objetos, describiendo los patrones de las superficies con funciones matemáticas; empleando el conjunto de funciones disponibles en el lenguaje, como bloques de construcción. A esta técnica se le conoce como creación de textura procedurales.

Geometría. Textura. Geometría con textura.

Fig. 2.2 Mapeo de textura sobre la superficie de un cono.

Usar texturas procedurales tiene algunas ventajas sobre el mapeo de texturas:
- El espacio para almacenarla es muy pequeño en comparación a una imagen.
- No tiene una resolución definida. En muchos casos proporciona una textura bien definida sin importar la distancia la distancia desde donde se le vea.
- No cubre un área especifica. Es decir, que puede extenderse de manera ilimitada para cubrir un área tan grande como se desee.

Pero también tiene algunas desventajas:
- Pueden ser difíciles de programar.
- En algunos casos aunque, se logren programar, es difícil controlar su comportamiento.
- Puede ser muy tardado evaluar una textura procedural, en comparación al tiempo de acceso de una imagen.

2.6 Texturas procedurales con patrones regulares.

Se pueden construir texturas cuyo patrón pueda repetirse una y otra vez sobre la superficie a intervalos regulares. En estos casos se puede utilizar la función mod(x, period) como bloque de construcción. Esta función regresa el residuo de la división x/period. Para construir la función periódica f(x) definida en el intervalo [0, p] basta con utilizar la expresión:
f(mod(x, p))

Algunas de las funciones que se pueden utilizar como bloques de construcción para patrones regulares son:

float step(float edge, x)
Regresa 0 si x < edge y 1 si x >= edge.

float smoothstep(float edge0, edge1, x)
Regresa 0 si x <= edge0, regresa 1 si x >= edge1, y realiza una interpolación entre 0 y 1 cuando edge0 < x < edge1. Esto es útil en donde se desea una función con una transición de valores suavizada.

type spline(float x; type val1, val2, ..., valn)
Tomando como valor la variable x contenida en el intervalo [0, 1], la función spline regresa el valor correspondiente a una interpolación cúbica entre los valores val1, val2,..., valn. La variable type puede ser del tipo: float, color, point o vector.

La figura 2.3 muestra el ejemplo de un "shader" de superficie que genera la textura de una pared formada por ladrillos. El "shader" se aplicó sobre un polígono. La forma de los ladrillos se construye con dos funciones rectangulares periódicas, una en sentido horizontal y otra en sentido vertical.

```
#define ANCHOLADRILLO      0.25
#define ALTOLADRILLO       0.08
#define ANCHOMEZCLA 0.01

#define ANCHOLM              (ANCHOLADRILLO+ANCHOMEZCLA)
#define ALTOLM             (ALTOLADRILLO+ANCHOMEZCLA)
#define FANM          (ANCHOMEZCLA*0.5/ANCHOLM)
#define FALM          (ANCHOMEZCLA*0.5/ALTOLM)

surface ladrillo( uniform float Ka = 1; uniform float Kd = 1;
        uniform color Cladrillo = color (0.5, 0.15, 0.14);
        uniform color Cmezcla = color (0.5, 0.5, 0.5); )
{
        color Ct;
        point Nf;
        float ss, tt, sladrillo, tladrillo, w, h;
        float scoord = s;
        float tcoord = t;

        Nf = normalize(faceforward(N, I));

        ss = scoord / ANCHOLM;
        tt = tcoord / ALTOLM;

        if (mod(tt*0.5, 1) > 0.5)
                ss += 0.5;

        sladrillo = floor(ss);
        tladrillo = floor(tt);
        ss -= sladrillo;
        tt -= tladrillo;

        w = step(FANM, ss) - step(1-FANM, ss);
        h = step(FALM, tt) - step(1-FALM, tt);

        Ct = mix(Cmezcla, Cladrillo, w*h);

        Oi = Os;
        Ci = Os * Ct * (Ka * ambient() + Kd * diffuse(Nf));
}
```

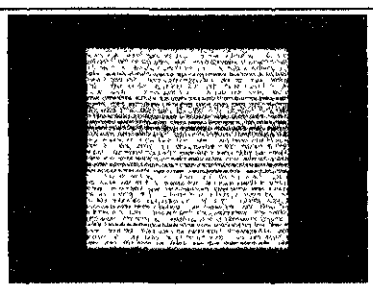


Fig 2.3 "Shader" que define la superficie de una pared con ladrillos. Se utilizan dos funciones cuadradas periódicas para definir los ladrillos.

2.7 Texturas procedurales con patrones irregulares.

Para generar este tipo de texturas, necesitamos una función que genere valores irregulares. Ken Perlin presentó, en 1985, la propuesta de una primitiva llamada noise() descrita en [22] y [23]. noise() es una función pseudoaleatoria que tiene las siguientes propiedades:

- es repetible, en el sentido de que regresara el mismo valor cuando se le llama en diferentes ocasiones con el mismo argumento.
- esta definida en los dominios de 1, 2, 3 o 4 dimensiones:
  - noise(float)
  - noise(float, float)
  - noise(point) /* también se puede definir con un argumento vector o normal. */
  - noise(point, float)
- El valor que devuelve noise() esta contenido en el rango [0, 1], con un valor promedio de 0.5. Si el valor de las coordenadas son valores enteros el valor que regresa es exactamente 0.5. Los valores que devuelve noise() tienden a estar dentro del rango [0.3, 0.7].
- noise() es una función de banda limitada. Sus frecuencias mas altas están contenidas en el rango [0.5, 1].
- noise() es isotrópica, es decir que no tiene dirección predilecta.
- no exhibe periodicidad o patrones regulares. En realidad es una función periódica, pero su periodo es tan grande que en raras ocasiones se tiene que considerar.

La función noise() tiene muchas variantes dependiendo del tipo de datos que esta regresa. Existe una función para cada uno de los siguientes tipos: float, point, vector y color. La función que debe de utilizarse, la determina el programador por medio de la operación de casting. Si no se indica el tipo con casting, el compilador trata de deducir que función se usará. Sin embargo esta práctica no es recomendable ya que el código puede presentar ambigüedades que causarían resultados impredecibles.

La figura 3.4 muestra el ejemplo de un "shader" de superficie que genera una textura con patrón irregular. El "shader" se aplica sobre un polígono. En este "shader" se evalúa la función noise() en cada punto de la superficie. frecuencias. La amplitud de la función se disminuye conforme la frecuencia aumenta. En este caso se evalúan cuatro valores de frecuencia distintos. Los valores resultantes se suman. Finalmente, el total de la suma se interpola con una función spline para asignarle un tono de azul al punto sobre la superficie. El resultado es una apariencia similar al mármol.

```
#define snoise(x) (2*noise(x) - 1)

#define AZUL_PALIDO     color(0.25, 0.25, 0.35)
#define AZUL_MEDIO      color(0.10, 0.10, 0.30)
#define AZUL_FUERTE     color(0.05, 0.05, 0.26)
#define AZUL_OBSCURO    color(0.03, 0.03, 0.20)
#define NNOISE          4

color color_marmol(float m) {
      return color spline( clamp(2*m+0.75, 0, 1), AZUL_PALIDO, AZUL_PALIDO,
            AZUL_MEDIO, AZUL_MEDIO, AZUL_MEDIO, AZUL_PALIDO, AZUL_PALIDO,
            AZUL_FUERTE, AZUL_FUERTE, AZUL_OBSCURO, AZUL_OBSCURO,
            AZUL_PALIDO, AZUL_OBSCURO); }

surface marmol_azul (   uniform float Ka = 1; uniform float Kd = 0.8;
      uniform float Ks = 0.2; uniform float esctextura = 2.5;
      uniform float rugosidad = 0.1; )
{
      color Ct;
      point NN;
      point PP;
      float i, f, marmol;

      NN = normalize(faceforward(N, I));
      PP = transform("shader", P) * esctextura;

      marmol = 0; f = 1;
      for (i = 0; i < NNOISE; i += 1) {
            marmol += snoise(PP * f) / f;
            f *= 2.17;
      }
      Ct = color_marmol(marmol);

      Ci = Os * (Ct * (Ka * ambient() + Kd * diffuse(NN))
            + Ks * specular(NN, normalize(-I), rugosidad));
}
```

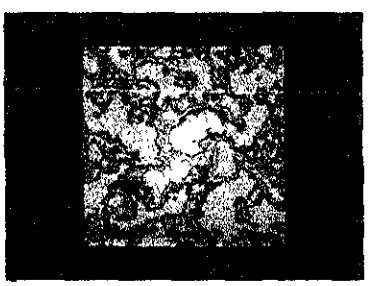


Fig. 3.4. "Shader" que define la superficie del mármol. Se utiliza una función noise() para generar las perturbaciones en las vetas.

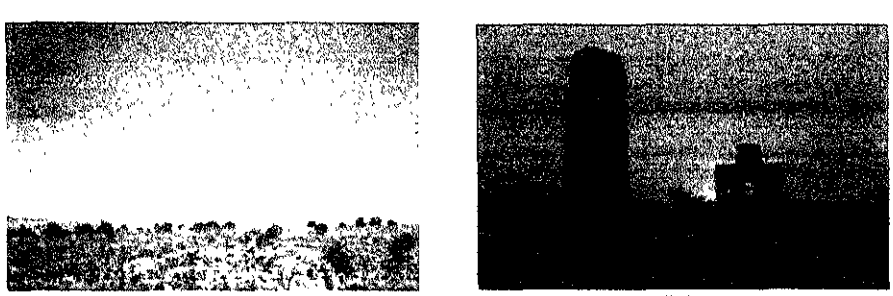
Capítulo 3.

Simulación de espacios abiertos.

La simulación de espacios abiertos requiere de técnicas adicionales a las que proporcionan los modelos de iluminación locales y globales. El principal problema, que no se resuelve en forma satisfactoria con estos modelos, es el de simular el color del cielo. A primera vista, se podría pensar que aplicando un azul uniforme en el fondo de todo ambiente virtual, bastaría para obtener un grado aceptable de realismo, sin embargo esto no es así. Cuando la luz solar penetra la atmósfera terrestre, la luz se dispersa hacia todas direcciones, provocando que todo el cielo se vea iluminado y de color azul. Sin embargo la dispersión no es uniforme, varía con la altitud, lo que produce que el color del cielo cerca del cenit es azul, y blanco cerca del horizonte.
Este capítulo tratará de explicar los fenómenos que producen los cambios de coloración en el cielo. También se revisará la propuesta de un modelo que permita simular la coloración del cielo, considerando las variaciones en la densidad atmosférica provocadas por la altitud y una simplificación del fenómeno de dispersión de Rayleigh[11].

3.1 Coloración del cielo.

El cielo no tiene un color uniforme a lo largo del día; la posición del sol es responsable de las variaciones en su tonalidad. Cuando el sol está en lo alto el cielo se vuelve azul, pero no presenta la misma tonalidad en todas direcciones; cerca del cenit es de un azul intenso y en el horizonte el color tiende hacia el blanco. Cuando amanece o anochece el color del horizonte cambia totalmente; éste se presenta con tonalidades entre el rojo y el amarillo.



(a) (b)

Fig. 3.1. (a) Imagen de un paisaje exterior, donde se aprecia el cambio de tonalidades en el azul del cielo. (b) Imagen de un paisaje exterior al atardecer, el cielo aparece con tonos rojos y amarillos.

Estos cambios se deben a un fenómeno conocido como dispersión. En su trayecto hacia la superficie terrestre la luz solar atraviesa la atmósfera, compuesta por una mezcla diversa de gases. En las capas exteriores predomina el ozono y más cercano a nosotros se encuentra el aire. El humo, el smog, el dióxido de carbono, los clorofluorocarbonos y en general todo tipo de contaminantes son producto de la actividad humana, y en las últimas décadas ha aumentado su presencia en la atmósfera.

Todos los gases absorben o dispersan en diferente medida la luz del sol. En la capa de ozono se absorbe una gran parte de los rayos ultravioleta, que no son visibles para los humanos. La absorción de luz visible en esta capa es tan pequeña que para efectos prácticos se puede ignorar. En cambio, las moléculas de aire provocan un fenómeno conocido como dispersión de Rayleigh; debido a que su tamaño es mucho menor a la longitud de onda de la luz solar. Las partículas como el polvo, humo, smog, cuyas moléculas son de mayor tamaño, presentan el fenómeno conocido como dispersión de Mie.

3.2 Dispersión de Rayleigh.

Cuando la luz incide sobre un material, los electrones en éste pueden absorber y reirradiar parte de esa luz. La extracción de energía de una onda incidente y la reemisión subsecuente de alguna porción de esa energía se conoce como dispersión.

En el caso de la atmósfera terrestre, el cielo es de un azul brillante y nos vemos rodeados de luz azul. La luz del sol que atraviesa la atmósfera, desde alguna dirección específica, es dispersada hacia todas direcciones por las moléculas de aire. Sin una atmósfera, el cielo de día sería tan negro como el espacio exterior. Con una atmósfera, el extremo rojo del espectro, en su mayor parte no se desviará mientras que el extremo azul o de alta frecuencias se dispersará substancialmente. Esta luz esparcida, llegará al observador desde muchas direcciones y el cielo entero aparecerá brillante y azul. Cuando el sol está muy bajo en el cielo, sus rayos pasan a través de un gran espesor de atmósfera. Los azules y los violetas son dispersados hacia los lados del haz luminoso con mayor fuerza que lo son los amarillos y los rojos que continúan propagándose a lo largo de una línea de observación desde el sol para formar las puestas de sol en la tierra.

Cuando la luz de diversas longitudes de onda (λ) inciden sobre una molécula de gas de diámetro d, donde $d << \lambda$, la intensidad relativa de la luz dispersada varía aproximadamente a una razón de λ^{-4}. Esta condición se satisface en la dispersión de la luz solar en la atmósfera terrestre, los átomos tienen un tamaño de aproximadamente una fracción de nanómetro mientras la luz visible tiene una longitud de 550 nanómetros. Es por ello que las longitudes de onda más cortas (luz azul) se dispersan con mayor eficiencia que las longitudes de onda más largas (luz roja). Por lo tanto, cuando la luz solar es dispersada por moléculas de gas en el aire, la radiación de longitud de onda más corta (parte azul) se dispersa más intensamente que la radiación de longitud de onda más larga (parte roja), y como resultado el cielo se ve azul. [27]. Lord Rayleigh fue el primero

en deducir esta dependencia, entre la frecuencia de la luz incidente y la intensidad relativa de luz dispersada: La intensidad relativa de luz dispersada es proporcional a la cuarta potencia de la frecuencia impulsora. A la dispersión de luz por objetos que son pequeños en comparación con la longitud de onda se le conoce, en honor a su descubridor, como dispersión de Rayleigh [17].

El motivo de porque la dispersión de la luz azul es mayor que la de la luz roja puede explicarse mediante una analogía mecánica. En un átomo o en una molécula, un electrón está ligado en su posición mediante fuerzas de restitución intensas. Tiene una frecuencia natural definida, de la misma manera que una pequeña masa suspendida en el espacio por medio de un conjunto de resortes. La frecuencia natural en los electrones de los átomos y de las moléculas está, por lo general, en una región que corresponde a la luz violeta o ultravioleta. Cuando se permite que caiga luz sobre tales electrones ligados, se crean oscilaciones forzadas a la frecuencia del haz de luz incidente. En los sistemas resonantes mecánicos es posible "excitar" al sistema más efectivamente cuando imprimimos sobre él una fuerza externa cuya frecuencia esté tan cerca como sea posible de la frecuencia natural de resonancia. En el caso de la luz, la frecuencia de la luz azul está más cerca de la frecuencia de resonancia natural del electrón ligado de lo que está la luz roja. Es de esperarse que la luz azul sea más efectiva en provocar que el electrón oscile, y que la luz azul se disperse más efectivamente [16].

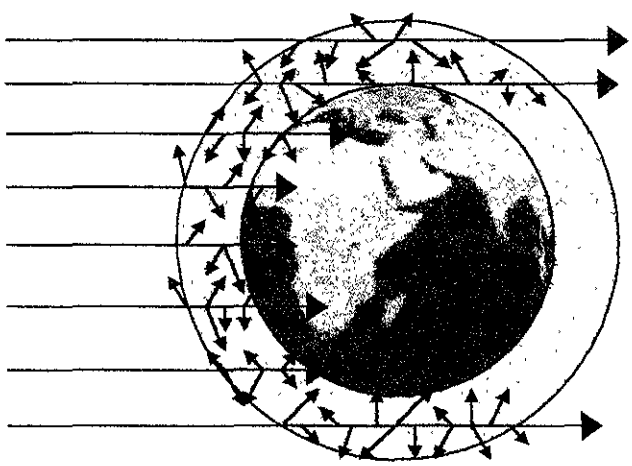


Fig. 3.2. Dispersión de la luz solar en la atmósfera terrestre.

Gustav Mie publicó en 1908 una solución del problema del esparcimiento para partículas esféricas homogéneas de cualquier tamaño. Es una solución complicada y tiene un gran valor práctico, particularmente cuando se aplica al estudio de suspensiones coloidales y metálicas, partículas interestelares, niebla, polvo, humo, la corona solar entre otros [17]. En el presente trabajo nos concentraremos en la coloración del cielo. Por lo que este fenómeno puede ser modelado adecuadamente, considerando únicamente la dispersión de Rayleigh.

3.3 Distribución de densidad atmosférica.

La eficiencia con que la luz solar presenta el fenómeno de dispersión es modificada por los cambios en la densidad de la atmósfera terrestre. La densidad de una sustancia se define como la relación entre su masa y la unidad de volumen. En nuestra atmósfera, la densidad presenta variaciones con respecto a la altitud: existe una mayor concentración de gases a bajas alturas, que va disminuyendo conforme nos elevamos hacia el espacio exterior; por ello la densidad al nivel mar es mucho mayor que la densidad en la cima del monte Everest.

Los modelos que describen estos cambios se les conoce como modelos de distribución de densidad atmosférica geométrica (GADD por sus siglas en inglés Geometric Atmospheric Density Distribution).

3.4 Modelos atmosféricos.

En este trabajo se considera un modelo atmosférico simplificado [11] basado en el cálculo de la distribución de densidad atmosférica, como elemento fundamental en la determinación del color del cielo.

Definimos la trayectoria óptica como el camino que recorre un rayo al atravesar la atmósfera. Conforme los rayos de luz recorren una trayectoria óptica, en su camino hacia la superficie terrestre, algo de esa luz se extingue y otra cantidad se adiciona debido a la dispersión u otros fenómenos. La suma de estos efectos describe el comportamiento de la atmósfera.

Los cambios que produce la atmósfera en la intensidad de los rayos de luz se puede describir mediante una ecuación diferencial:

$$dI = \sigma(x) + E(x)dx,$$

donde:
x es una posición en la atmósfera, expresada en coordenadas tridimensionales,
$\sigma(x)$ describe la extinción de la luz por unidad de longitud, en una trayectoria óptica.
$E(x)$ describe la dispersión o adición por unidad de longitud, en una trayectoria óptica.

Esta expresión nos lleva a dividir el modelo en dos partes:
- Un modelo de distribución de densidad atmosférica geométrica que describa a $\sigma(x)$.
- Un modelo de dispersión de Rayleigh que describa el término $E(x)$.

Dado que la densidad atmosférica es responsable de la atenuación de luz en la atmósfera terrestre, su distribución geométrica describe el término $\sigma(x)$ en la ecuación diferencial anterior. Al integrar $\sigma(x)$ sobre una trayectoria óptica, obtenemos:

$$\tau = \int_0^{t_e} \sigma(x)dt$$

que representa la profundidad óptica sobre la trayectoria. El parámetro t nos ayuda a localizar posiciones dentro de la trayectoria del rayo. Su valor esta contenido en el intervalo [0, t_e].

La ley de Beer establece que: "Cuando un rayo de luz monocromática pasa a través de un medio absorbente, su intensidad disminuye exponencialmente a medida que aumenta la densidad". Aplicando esta ley sobre la trayectoria óptica, podemos obtener su transparencia expresada en función de τ. La expresión resultantes es: $T=e^{-\tau}$

Para calcular la extinción total sobre la trayectoria óptica podemos hacer una simplificación utilizando el concepto de opacidad. La opacidad representa la proporción de luz que es absorbida al interactuar con un medio, sea una superficie o un volumen. Por lo general los valores para la opacidad están normalizados, es decir que están contenidos dentro del intervalo [0, 1]. Un medio transparente tiene una opacidad de 0 y un medio completamente opaco tiene una opacidad de 1. Dado que la T representa el nivel de transparencia en una trayectoria óptica, la extinción total se puede obtener al aplicarse T sobre un medio completamente opaco, es decir: 1-T. Esta simplificación nos permite obtener la extinción total en forma sencilla; no se trata del mejor modelo en términos de exactitud, pero produce buenos resultados con el mínimo número de operaciones.

Finalmente, la porción de intensidad que se extingue, se debe reemplazar con el color de la atmósfera, que hará la contribución del termino E(x) que aparece en la ecuación diferencial anterior. Esta contribución es proporcionada por la dispersión de Rayleigh que explicaremos más adelante.

3.4.1 Modelo de Distribución de Densidad Atmosférica.

La precisión que se requiere al representar la distribución espacial de una densidad atmosférica, debe estar en función de la escala en la que se utilizará. Los modelos más simples, con una densidad constante, son adecuados a pequeñas escalas en donde se aprecian porciones del cielo; por ejemplo la vista por una ventana desde un interior. Un modelo de escala intermedia tomará en cuenta los cambios de densidad con la altitud; estos son apropiados para paisajes de exteriores. Mientras que un modelo global deberá tomar en cuenta los cambios de la densidad, con la altitud y con la curvatura de la tierra. Con estos últimos, se podrá modelar el efecto de la atmósfera en imágenes donde se aprecia todo un planeta desde el espacio exterior.

El modelo con densidad constante tiene una solución muy sencilla. Integrando τ con $\sigma=c$, donde c es una constante, nos da:

$$\tau=ct_e$$

Un modelo de densidad intermedia muy útil para el "rendering" de paisajes, tiene una distribución de densidad que varia a una razón de e^{-z}, donde z es la altitud relativa al

plano horizontal. Esta relación se debe a que en la naturaleza, la densidad de las moléculas de aire decrece en forma exponencial con la altitud.

Para tener un mejor control en el comportamiento de este modelo, se pueden agregar dos términos más, y obtener la expresión:

$$\sigma=Ae^{-Bz}$$

donde:
A: controla la densidad total y
B: controla la disminución de la densidad con la altitud.

Esta función de densidad se puede integrar analíticamente. Primero observemos que, por la naturaleza de la densidad, la función esta expresada en términos de la coordenada z en lugar del parámetro t, por lo tanto se deben realizar algunos cambios en la expresión para integrar sobre $[z_0, z_e]$, que corresponden a la altitud del inicio y final del rayo respectivamente. Con este propósito, podemos escribir la ecuación como:

$$\tau = At_e \int_{z_0}^{z_e} e^{-Bz_d z}\, dz$$

donde:
z_0 es la coordenada z del origen del rayo.
z_e es la coordenada z del final del rayo.
z_d es la coordenada z del vector de dirección normalizado del rayo.

En esta nueva expresión, el término z_d modula el cambio en z debido al vector de dirección del rayo de luz y t_e modula la densidad total considerando la magnitud de la trayectoria óptica.

Al resolver la integral obtenemos:

$$\tau = \frac{At_e}{Bz_d}(e^{-z_e} - e^{-z_0})$$

Para obtener el resultado en términos de la altitud inicial del rayo y el vector de dirección, podemos sustituir $z_e=z_0+z_dt_e$ en la expresión anterior, lo que nos da:

$$\tau = \frac{At_e}{Bz_d}(e^{-(z_0+z_dt_e)} - e^{-z_0})$$

Esta expresión tiene un inconveniente: es indeterminada cuando $z_d=0$, es decir para rayos que son horizontales. Cuando se presenta este caso, se emplea una simplificación de la expresión anterior, para evitar la posible división por cero:

$$\tau = At_e e^{-Bz_0}$$

Un modelo global, además de la altitud, debe de considerar la curvatura de la tierra. Por lo tanto la distribución de densidad en este caso será de tipo radial, es decir que varía exponencialmente con un radio que describe las variaciones en la distribución desde un punto central.

En este modelo el radio r de la distribución, se relaciona con la posición t del rayo con la expresión:

$$r(t) = \sqrt{\alpha^2 + 2\beta t + t^2}$$

donde α y β son constantes determinadas por la trayectoria del rayo y el origen o centro de la distribución de densidad radial. Al calcular τ en función de σ se tiene que sustituir la expresión:

$$\tau = \int \sigma(t)dt \quad \text{con } \tau = \int \sigma(\sqrt{\alpha^2 + 2\beta t + t^2})dt$$

La función de distribución más sencilla y precisa es $\sigma=e^{-\tau}$ que al sustituirse en la expresión anterior nos da:

$$\tau = \int_0^{te} Ae^{-B(\sqrt{\alpha^2+2\beta t+t^2})}dt$$

Esta expresión requiere un método numérico para resolverse.

3.4.2 Una aproximación a la dispersión de Rayleigh.

Dado que la dispersión de Rayleigh es la causante de la coloración del cielo, debemos emplear un modelo que genere el mismo efecto en las imágenes.

La dispersión de Rayleigh adiciona luz azul a lo largo de la trayectoria óptica; esto causa que el cielo se vea azul. Por otra parte, la dispersión desplegada a lo largo de la trayectoria óptica enrojece la luz que viene del fondo causando que los amaneceres y atardeceres sean rojos.

Hasta ahora hemos tratado a τ, como si fuera un valor escalar. En la dispersión que depende de la longitud de onda de la luz λ, la τ debe de manejarse como un vector sobre λ, y cada muestra de λ debe tener un valor independiente de τ. Por lo tanto cada componente del vector τ requiere una evaluación separada de la expresión de la ley de Beer, es decir: e^{-t_λ}. Para determinar el número de componentes que debe tener τ, consideramos la naturaleza de nuestra percepción al color. Los humanos somos capaces de recrear toda la gama de colores, mezclando solo tres de ellos: rojo, verde y azul. Estos colores primarios, son los mismos que se utilizan en televisores y monitores de computadoras para generar imágenes en color. Si expandemos τ a un vector "rgb", seremos capaces de generar una gama extensa de colores que cambiarán en función de la distribución de densidad atmosférica. Para obtener el vector "rgb", deberemos calcular una muestra para el rojo, una distinta para el verde y otra para el azul. Esto se logra aplicando el vector $(e^{-\tau}, e^{-2.25\tau}, e^{-21\tau})$ a la distribución de densidad τ. Con esta expresión obtenemos el azul característico del cielo. Los factores que multiplican a τ en cada elemento del vector son responsables de modular el valor de la densidad en el componente del vector de color correspondiente.

Adicionalmente se debe considerar la iluminación producida por los rayos del luz al incidir en la superficie que modela el cielo. Esta iluminación constituye otra contribución de color.

Este método consigue una aproximación computacionalmente sencilla y con una buena aproximación al fenómeno de dispersión de Rayleigh.

3.4.3 Cuadratura trapezoidal para la función de densidad $\sigma=e^{-r}$.

Para calcular densidades atmosféricas que no son integrables analíticamente, se propone un método de integración numérica. Se trata de un método de cuadratura trapezoidal adaptativo.

El modelo atmosférico global más sencillo y preciso, donde se considere la curvatura de la tierra, es:

$$\sigma(r) = Ae^{-Br}$$

donde:
A modula la densidad al nivel del mar,
B es el coeficiente modula la altitud, y
r es la altura sobre el nivel del mar.

Dado que la distribución atmosférica está descrita en forma exponencial, es de esperarse que la densidad y su razón de cambio sean más grandes cerca de la superficie terrestre. Aprovechando esta característica podemos utilizar cuadratura con incrementos adaptativos para evaluar la integral. El tamaño del incremento será inversamente proporcional a la magnitud local de σ. De esta manera, donde la densidad y su razón de cambio son altas, el tamaño del incremento es pequeño; y donde la densidad es baja, el tamaño del incremento es relativamente largo. Para acelerar aún más el proceso de rendering, se emplea una prueba para rechazar si los rayos nunca llegan a estar dentro de una distancia mínima al centro de la densidad, regresando cero cuando la integral tiende a ser muy pequeña. .

Aplicar integración trapezoidal, implica que la variación entre las muestras de densidad será de tipo lineal. Por lo tanto la trayectoria óptica de un cualquier intervalo estará dada por

$$\tau=\Delta/2(\sigma_i+\sigma_{i-1})$$

donde
Δ es el tamaño del incremento.
σ_i es el valor de la densidad actual.
σ_{i-1} es el valor de la densidad anterior.

Aplicando el concepto de opacidad podemos obtener la extinción diferencial, que estará dada por la expresión:

$$dO = 1-e^{-\tau}$$

y la dispersión estará dada por:

$$dC = I(1 - (e^{-\tau}, e^{-2.25\tau}, e^{-21\tau}))$$

donde I es la intensidad de iluminación directa en el punto de muestreo.

Estos valores diferenciales se suman conforme se recorre la trayectoria óptica. Finalmente para implementar el recorrido por el rayo, se emplea una técnica conocida como Ray Marching.

3.5 Técnica de Ray Marching.

La técnica de "ray marchig" es capaz de generar efectos atmosféricos muy sofisticados. Consiste en recorrer los rayos que se forman desde la posición del observador, hasta la superficie de los objetos, tomando muestras a lo largo del rayo y calculando la extinción atmosférica en cada una de dichas muestras. Los rayos están representados por los vectores incidentes a las superficies.

El pseudocódigo básico del algoritmo, se muestra en la figura 3.3.

```
Selecciona un incremento apropiado para recorrer el rayo.
longitudI = length(I)
Pactual = P-I
while (longitudI > 0)
        toma la muestra de la densidad y la iluminación en Pactual
        Cvol += (1-Ovol) + incremento*(luz dispersada)
        Ovol += (1-Ovol) + incremento*(densidad local)
        Pactual += incremento*normalize(I)
        longitudI -= incremento
}
Pone los valores Ci/Oi para la superficie
```

donde:
P es la posición del punto sobre la superficie donde incide el rayo.
I es el vector incidente.
longitudI es la longitud del rayo o vector incidente.
Pactual es el punto de muestra donde se evalúa densidad y opacidad.
Cvol, Ovol son el color y la opacidad del punto de muestra respectivamente.
Ci y Oi son el color y la opacidad total debida al volumen.

Fig. 3.3 Pseudocódigo básico del algoritmo de "Ray Marchig".

En el pseudocódigo podemos apreciar que, primero se toman puntos de muestra, que están separados entre sí por la distancia definida en la variable incremento. Después se calcula la densidad e iluminación en cada uno de los puntos, para sumarlos y obtener al final el color y la opacidad producida por todo el volumen.

Al aplicar un algoritmo de integración trapezoidal, este algoritmo sufre una pequeña modificación. Primero, se divide el volumen que atraviesa el rayo en segmentos. Después, para cada segmento se calculan las densidades e iluminación en los puntos inicial y final. Por último, se promedian los dos valores obtenidos, para calcular el valor de densidad e iluminación media, para cada segmento.

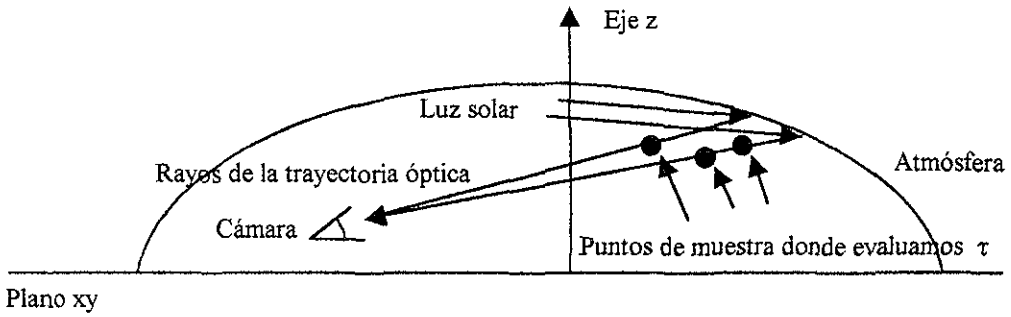


Fig. 3.4. Proceso de Ray Marching aplicado en el modelo atmosférico.

3.6 Generación de nubes.

Dado que es muy raro encontrar días claros en los no aparezca una sola nube en el cielo, debemos considerar un método para la generación de nubes.

Modelar fenómenos naturales, como las nubes, es una tarea difícil. Constantemente se proponen nuevos métodos para generar imágenes donde estos fenómenos se presenten. Una clase muy general de técnicas usadas para este propósito, son las de modelado y creación de texturas procedurales volumétricas. Los modelos volumétricos procedurales emplean funciones de densidad tridimensionales (fdv(x, y, z)) que definen la densidad de un espacio tridimensional continuo. Las funciones de densidad se usan ampliamente en graficación por computadora para modelar y animar gases, fuego, líquidos, nubes y otros objetos.

Las texturas procedurales volumétricas, también conocidas como texturas sólidas, son espacios de color tridimensionales que rodean al objeto. Cuando una textura sólida se aplica sobre un objeto, el efecto obtenido es similar a ver el espacio sólido, esculpido con la forma del objeto; en otras palabras, es como si realizáramos esculturas con bloques definidos por el espacio sólido. Estas texturas tienen la desventaja de no ser un modelo realmente tridimensional, porque sólo consideran la superficie del objeto aunque el procedimiento de textura sólida es definido en un espacio de tres dimensiones. Para remediar este inconveniente, se puede extender el modelo para que la función defina la densidad considerando volúmenes tridimensionales, utilizando para ello la técnica de "ray marching".

Además del color, los espacios tridimensionales también pueden aplicarse sobre otras características del objeto, como la transparencia o la misma geometría. Cuando se

generaliza el concepto para englobar todas esas características, los espacios reciben el nombre de espacios sólidos.

Los espacios sólidos son espacios tridimensionales asociados a un objeto con la finalidad de controlar alguno de sus atributos. Los atributos pueden ser el color, la transparencia, la geometría, la rugosidad, la reflectividad, la transparencia, las características de iluminación y el sombreado de objetos. Las hipertexturas y las funciones de densidad de volumen son espacios sólidos donde se manipula la geometría.

Los espacios sólidos pueden describirse de forma sencilla en términos matemáticos. Se pueden considerar como una función de un espacio tridimensional a un espacio n-dimensional, donde n puede ser cualquier entero positivo distinto de cero:

$$S(x,y,z) = F, F \in R^n, n \in 1,2,3,...$$

La definición de un espacio sólido puede cambiar con el tiempo, por tanto, el tiempo se puede considerar como una cuarta dimensión en la función del espacio sólido. Para la mayoría de los casos, S es una función continua en el espacio tridimensional.

En el presente trabajo utilizamos el método de generación de nubes propuesto en [2][11]. En este modelo, se requieren dos elementos para definir una nube:

1) La superficie de una figura geométrica que define la macroestructura de la nube, es decir su contorno.
2) La evaluación de una función de densidad contenida dentro de un volumen, que forme la microestructura de la nube, lo que le dará su apariencia nebulosa.

La idea básica de este método es formar nubes con ayuda de un espacio sólido. La superficie de una figura geométrica define la forma del volumen, y una función de densidad determina la forma gaseosa que tendrá el interior de la nube. La función de densidad se obtiene a partir del espacio sólido. Para considerar todo el interior de la nube, se evalúa la función de densidad sobre un volumen utilizando la técnica de "ray marching". El volumen, que define el espacio sólido, se construye de la siguiente forma:

1) Se trazan rayos, desde la posición de la cámara hacia, cada uno de los puntos sobre la superficie que define el contorno de la nube.
2) El punto final en la trayectoria de cada rayo está dado por el punto donde incide el rayo sobre la superficie.
3) Para determinar el punto inicial se utiliza la expresión para resolver ecuaciones cuadráticas. Ambos puntos, inicial y final definen el volumen que recorrerá cada rayo, sobre el cual se evaluará la función de densidad por medio de "ray marching".

Para darle una apariencia real a la nube, el espacio sólido debe presentar un patrón irregular, no periódico. Esto se consigue con una de las aplicaciones de la primitiva noise(): la función "fractional Browmanian noise()", cuyo nombre abreviado es "fBm" [2]. La idea básica en "fBm" es sumar varias copias de la primitiva noise() a diferentes frecuencias. A mayor frecuencia le corresponderá menor amplitud. La función presenta

militud a diferentes escalas, es decir que se suman diferentes copias de sí misma a ferentes escalas. Esto genera un patrón agradable, complejo, con apariencia natural ue imita muchas cosas de la naturaleza. Con la función "fBm" es posible generar una nplia gama de fractales. Los fractales se pueden describir en forma sencilla y heurística omo objetos complejos, cuya complejidad es producida por la repetición de formas obre un rango de escalas.

n "fBm" la relación que existe entre la cantidad con la que escalamos la amplitud con especto a la cantidad con que escalamos la frecuencia determina la dimensión fractal de a función. La dimensión fractal es una propiedad muy peculiar en los fractales. Cuando ablamos de cuerpos geométricos inmediatamente pensamos en dimensiones uclidianas, que tienen valores enteros: una dimensión de cero corresponde a un punto, no a una línea, dos a un plano, y tres al espacio. Los fractales, en cambio, poseen limensiones de valores reales como 2.3, que representan la complejidad visual de la onstrucción fractal.
El carácter esencialmente fractal de la función "fBm" nos permite usar modelos fractales para representar nubes. La desventaja que tiene esta función es que el costo para calcularla es alto, debido a que se invoca en muchas ocasiones a la función noise().

Si las nubes que se desean producir son delgadas y no requieren de cálculos de sombra, el proceso que hemos descrito es suficiente para crearlas. Pero si se desea aumentar la calidad de las nubes se debe considerar el manejo de sombras. El costo de cálculo se incrementa, pero si se tienen los medios, los resultados son mejores. El sombreado de las nubes se consigue trazando un rayo de sombra hacia la fuente de luz para cada muestra. Cada rayo de sombra se recorre desde el punto de muestreo a la posición de la fuente de luz, calculando la opacidad total a lo largo del rayo de sombra y usando esto como un multiplicador para la luz total alcanzada en el punto de muestreo.

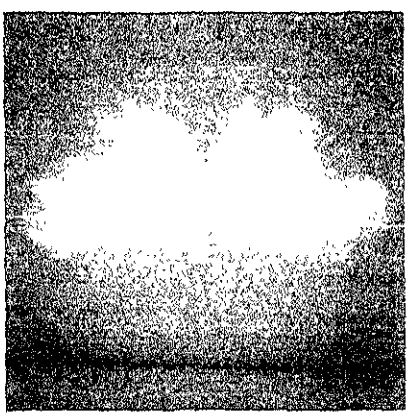


Fig. 3.5 Nube generada con un espacio sólido. La forma es modelada con varias esferas.

Capítulo 4.

Aplicación del modelo atmosférico en un ambiente virtual.

Para demostrar la efectividad del modelo, se generó la animación de un ambiente virtual. El ambiente generado muestra una posible área de aplicación de la propuesta.
Este capítulo describe las etapas realizadas en la creación de la animación.

4.1 Arquitectura de un prototipo para generar animaciones.

Para generar una animación con *BMRT* o *PRMan* se requiere del uso de diferentes herramientas de software: modeladores, programas en C o C++ y "shaders". La arquitectura mostrada en la Fig. 4.1 integra todos estos elementos.

Rhinoceros fue el modelador que se utilizó para crear la geometría del ambiente virtual. Con él se generaron los archivos en formato RIB. *Rhinoceros* no ofrece la posibilidad de crear animaciones. Todos los archivos que genera presentan solamente una vista del ambiente, por ello se hizo necesario hacer un programa que convirtiera la geometría de formato RIB a C. Expresando la geometría en código C aprovechamos la capacidad de la API *RenderMan* de generar varias vistas de un ambiente.

La API de *RenderMan* esta diseñada para ser utilizada con C. Sin embargo es posible integrarla a un paradigma de programación orientada a objetos en C++. El convertidor, el modelo atmosférico y la cámara fueron programados como clases.
La clase convertidor se utiliza en un programa ejecutable por separado y su única tarea es realizar la conversión.

El modelo atmosférico fue programado como un "shader"; para su uso en un ambiente virtual, se requiere de una figura geométrica que represente el espacio que ocupa la atmósfera terrestre. La clase BovedaCeleste crea una semiesfera, declara los parámetros que requiere el "shader" para que se aplique correctamente sobre ella.
La clase Camara, traslada la cámara a la posición definida por el usuario y apunta la lente hacia la orientación deseada. El programa generador de imágenes crea una instancia de la clase BovedaCeleste y una de Camara para integrarlas a la geometría del ambiente. Este programa es el responsable de generar las imágenes que formarán parte de la animación.

Finalmente para generar la animación empleamos el paquete de software *Paint Shop* que crea un archivo en formato AVI.

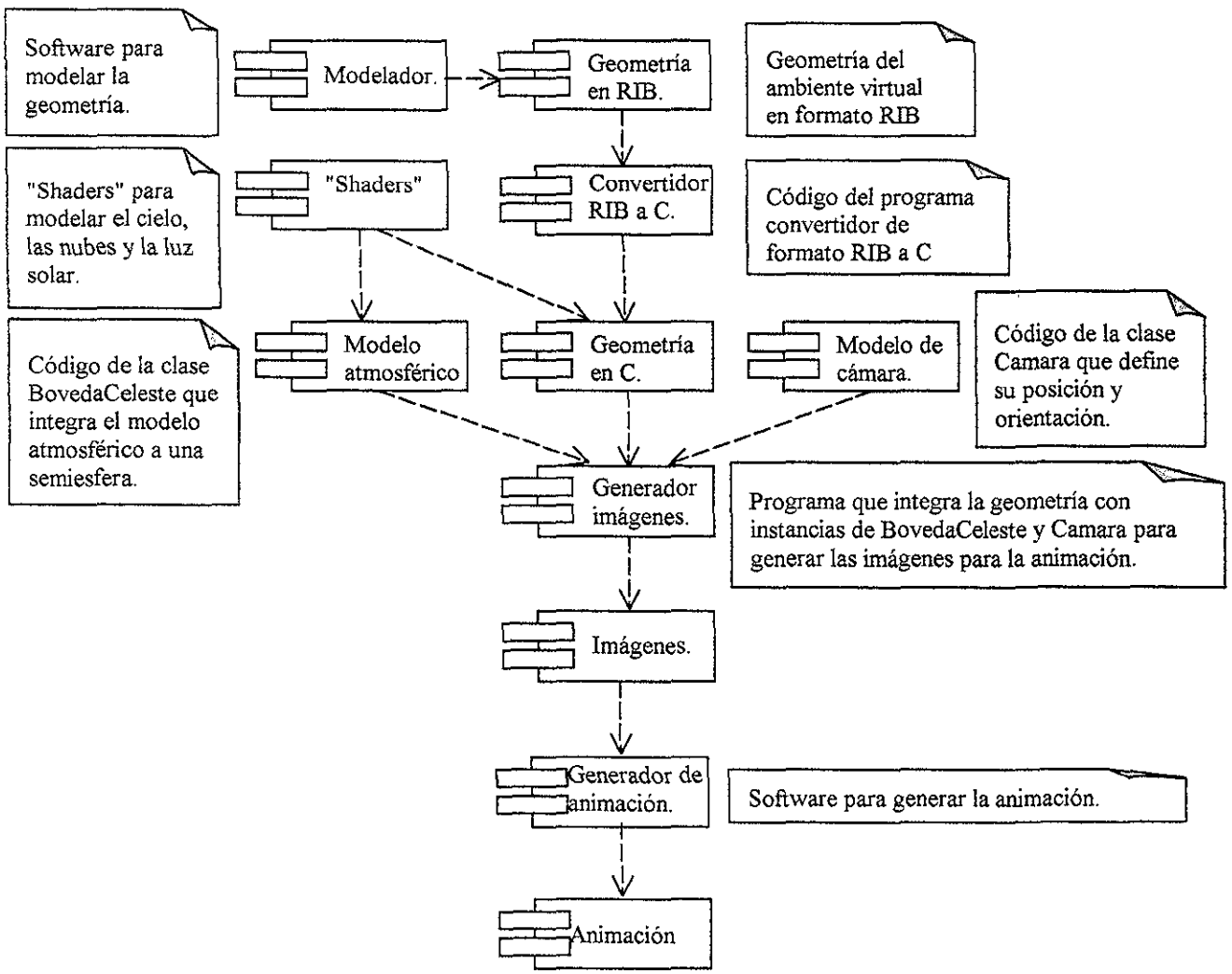
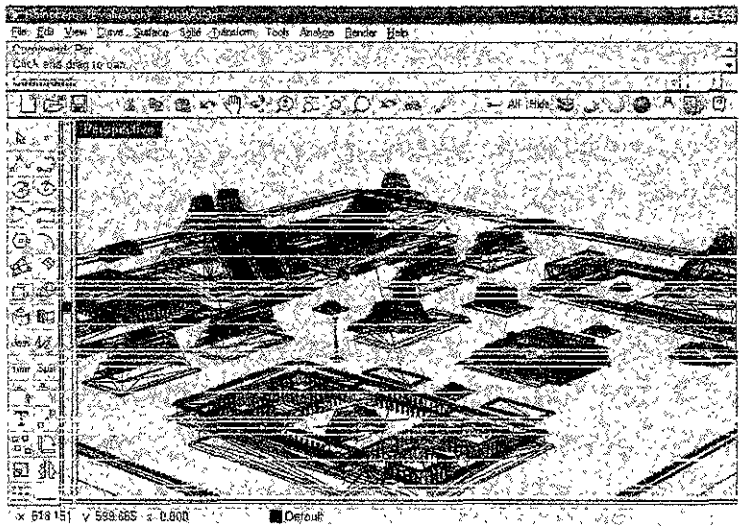


Fig. 4.1 Diagrama de componentes del prototipo.

4.2 Geometría del ambiente virtual.

El modelo del ambiente virtual consiste de 56585 vértices, 87006 polígonos, agrupados en 546 mallas poligonales.



ig 4.2 Geometría del modelo en Rhinoceros.

l modelo original estaba orientado en tal forma, que la superficie base de la geometría era l plano xz. El modelo atmosférico se diseño pensando en que el eje que determinaría la ltitud fuera z, en lugar de y. Por este motivo se cambió la orientación de todas las figuras eométricas para que la superficie base fuera el plano xy.

osteriormente, se creó una descripción del modelo en formato RIB de *RenderMan*. *Rhinoceros* permite importarla en este formato. El archivo que genera, contiene solo un uadro del ambiente, que es la vista mostrada en la ventana de la perspectiva activa en el nodelador. Después este archivo se modifica, empleando un editor de texto, para agregar as llamadas a los "shaders" que definen las texturas en las superficies de los polígonos. En articular, usamos un "shader" para superficies mate en todos los polígonos. Este "shader" escribe un modelo de iluminación puramente difuso, adecuado para paredes recubiertas on estuco.

ig. 4.3 Dos vistas diferentes de la geometría del ambiente virtual. Se emplearon los shaders" del modelo atmosférico y el de la luz proveniente del sol. Las imágenes se eneraron en BMRT.

ara representar las nubes se agregan varias esferas y sobre ellas se aplica el "shader" orrespondiente. La principal fuente de iluminación estará dada por el sol por ello se declara l uso del "shader" que simula esta fuente de luz.

Una característica que ayuda a realzar el realismo en las imágenes es el uso de sombras. Como se mencionó en el primer capítulo, *BMRT* utiliza el algoritmo de "ray tracing" para formar imágenes. Este algoritmo también permite determinar las sombras. Por omisión *BMRT* no las genera; se tiene que indicar explícitamente, empleando la siguiente instrucción:

Expresada en RIB Attribute "light" "shadows" ["on"]
Expresada en C RiAttribute("light", "shadows", &encendido, RI_NULL);

Esta línea se debe declarar inmediatamente antes del "shader" de fuente de luz con el que determinará las sombras. En *BMRT* cada objeto puede especificar si se incluye o se ignora en el proceso de determinación de sombras. Esto es muy útil para acelerar el cálculo de la imagen. La semiesfera, que determina la bóveda celeste, y el polígono, que representa el piso, no requieren considerarse en el proceso de generación de sombras, porque son elementos que no se proyectan sobre otras superficies.

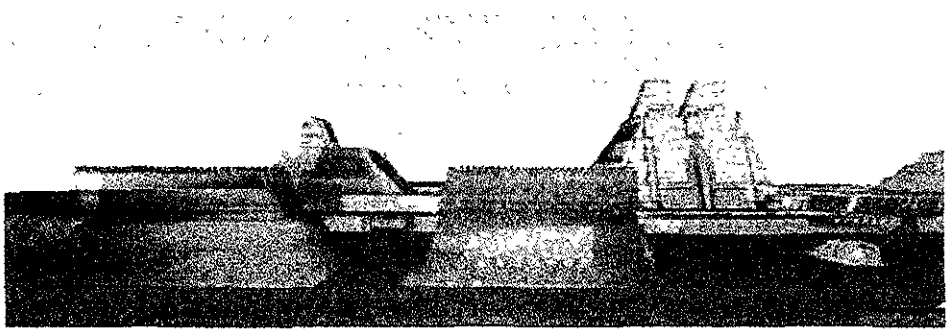


Fig. 4.4 *Imagen que muestra la generación de sombras en el ambiente*

4.3 Conversión del modelo de formato RIB a C.

Rhinoceros tiene la capacidad de generar archivos en formato RIB, que describen solo un cuadro de la escena. No es posible generar archivos que definan una secuencia de imágenes. Si deseamos generar una animación utilizando exclusivamente este modelador, se deben generar muchos cuadros, cada uno con una posición del observador diferente. Almacenar, en formato RIB, estos cuadros es costoso; cada uno mide 3.66 MB. Por lo tanto se debe de encontrar una alternativa, que no requiera el guardado de grandes volúmenes de información. Explotando la capacidad de la API de *RenderMan*, de generar varios cuadros por medio de un programa escrito en C, podemos evitar el problema de almacenamiento. Por ello, se escribió un programa en C++ que hiciera la traducción de formato RIB a su equivalente escrito en lenguaje C. Terminado el programa convertidor, se tradujo toda la geometría.

En las versiones más recientes de *BMRT*, se proporciona un mecanismo para incluir archivos completos RIB en código RIB o C. Este mecanismo fue muy conveniente para ahorrar los tiempos de compilación de código. En nuestro ambiente virtual los edificios no cambian, ni de forma ni de posición, en los diferentes cuadros de la animación. Son, por

lamarlos de alguna forma, la parte estática del ambiente. Su complemento, la parte dinámica, son los elementos que sufren cambios a través de los diferentes cuadros. Estos elementos son la bóveda celeste, las nubes y el sol. Sólo la parte dinámica requiere pasar, en forma repetida, por el proceso de compilación, La parte estática se conserva en RIB y se ncluye desde el programa en C. Con esta división de elementos, se reduce el proceso de raducción. Sólo la parte dinámica tiene que pasar por este proceso.

La instrucción que permite la inclusión de un archivo RIB desde C es:
RiReadArchive("geometria.rib", RI_NULL);

El anexo 4 muestra el código completo del traductor RIB a C. El anexo 5 muestra el programa traducido a C.

4.4 Programación del modelo atmosférico empleando el lenguaje de "shading".

Para desarrollar el modelo atmosférico basado en el capítulo anterior, nos basamos en un "shader" volumétrico con tres diferentes modelos de densidad atmosférica, relacionados con la escala a la que se utilizarán. Se trata de un "shader" muy general que maneja esta diversidad de modelos. Para cumplir su objetivo, la resolución de la integral, que determina la densidad, se resuelve, en los tres casos, con el método de cuadratura trapezoidal. Sin embargo este método no es adecuado cuando el "shader" se aplica en ambientes de escala pequeña o intermedia, porque se invierte mucho tiempo en la resolución de la integral, aumentando el tiempo de generación de la imagen. Para hacer más eficiente el proceso que determina la coloración del cielo, se programó un "shader" que está diseñado específicamente para escalas intermedias. Esta consideración es suficiente en la simulación de ambientes abiertos, porque las variaciones en la tonalidad del cielo varían sólo con la altitud. La diferencia, con respecto al primer "shader" mencionado, es que la integral se resuelve en forma analítica en lugar de hacerlo con un método numérico.

Para tener un punto de comparación entre ambos esquemas, se programaron los dos tipos de "shaders". Ambos consideran exlusivamente ambientes de escala intermedia. La mejora en tiempo de procesamiento, del "shader", donde se resuelve la integral en forma analítica, es considerable. Al final del capítulo mostraremos los tiempos de cálculo de algunas imágenes del prototipo.

Para simular la luz solar, que es la principal fuente de iluminación en el ambiente, se programó un "shader" de fuente de luz. La sentencia de control "solar" fue el bloque de construcción básico. Los rayos del sol son tratados como vectores paralelos; con esto se da la impresión de que la fuente de donde proviene se encuentra a una distancia muy lejana. El "shader" recibe como parámetro la posición donde se encuentra el sol para darles la dirección correcta a los rayos.

ara generar nubes se debe de contar con dos elementos, mencionados al final del capítulo nterior:

) Las macroestructuras que definen el contorno de las nubes, se modelan con esferas.
) La microestructura que define la apariencia de la nubes se programa como un "shader" de superficie, cuya tarea es modelar su forma gaseosa, evaluando la función de densidad en el interior de un volumen. Este "shader" se aplica sobre las esferas del punto anterior.

os códigos completos de estos "shaders" se puede revisar en el anexo 3 del presente abajo.

os "shaders" incluyen parámetros que ayudan a modificar su comportamiento, al utilizarse n la creación de secuencias de imágenes para animación.
ara el "shader" del modelo atmosférico, el cambio que debe percibirse, en los diferentes uadros, es el color del cielo. Para lograrlo se tienen varias opciones. En primer lugar el shader" tiene la posibilidad de considerar o ignorar, la iluminación que se aplica sobre el mbiente, para determinar el color del cielo. El parámetro iluminación se encarga de esta area; si es 0 no toma en cuenta la iluminación del ambiente y el usuario tendrá que roporcionar el color que se aplicará, asignándolo al parámetro colorilum. El valor por misión es blanco. Si al parámetro iluminación se le asigna un 1 se toma en cuenta el mbiente y se ignora el valor proporcionado en colorilum; este último es el comportamiento or omisión. Para la secuencia que creamos en todos los cuadros se considera la uminación del ambiente. El color de la bóveda también juega un papel importante en la leterminación del color presente en el cielo. Se utilizó un color gris [0.2 0.2 0.2] para la óveda, el color azul es proporcionado por la simulación de la dispersión de Rayleigh. Para as puestas de sol y los amaneceres este valor se modifica para realzar los rojos y amarillos. l parámetro de densidad es un factor que multiplica los valores calculados en la función de lensidad atmosférica. El parámetro decremento modifica el exponente en la determinación le la función densidad. Estos dos parámetros nos pueden ser de utilidad para modificar el olor del cielo. A mayor densidad el cielo se vuelve más oscuro y a menor densidad se uelve más claro. Mientras que decremento nos permite controlar el grado de cambio entre os diferentes matices de color en la atmósfera, un valor de 50 proporciona un cambio suave natural.

ara el "shader" de iluminación de la luz solar los tres parámetros que controlan su omportamiento en la animación son la intensidad, que modifica la cantidad de iluminación roporcionada; lightcolor, que determina el color de la iluminación; y sol que es un vector que ndica la dirección de los rayos solares. En la animación la intensidad se mantuvo constante, l color de la iluminación cambio ligeramente al amanecer y al atardecer agregando más rojo amarillo. En el resto del día se uso blanco. El vector de la posición del sol cambió para imular desde su salida hasta su ocaso, siguiendo una trayectoria recta. Las sombras se lefinen con respecto a la luz solar, por lo tanto cuando haya cambios en la posición del sol, utomáticamente las sombras se modificarán ajustándose a las nuevas condiciones de uminación.

ara el "shader" de nubes el parámetro que controla la animación en las nubes es un vector amado movimiento. Su función es modificar el espacio sólido donde se evalúa la función de lensidad de la nube. Esta técnica se emplea muy a menudo en graficación. La referencia

11] expone esta técnica para animar gases. La modificación consiste en sumar el vector al unto donde se evalúa la función noise(), con esta sencilla operación se recorre el espacio ólido una distancia equivalente a la magnitud del vector siguiendo la dirección que este ndica.

4.5 Modelo de cámara.

El modelo de cámara nos permite cambiar la posición del observador y la dirección hacia onde está mirando. Controlando estas dos propiedades podemos obtener imágenes de ualquier lugar contenido en el ambiente.
En *RenderMan*, la imagen de una escena se forma desde un punto de vista particular. Este unto es el lugar donde se coloca una cámara virtual. Para facilitar el proceso de generación e imágenes, la cámara se define en el origen de un sistema de coordenadas propio, onocido como espacio de cámara, y está mira siempre hacia el eje positivo z. En toda scena, por omisión, el sistema de coordenadas de cámara coincide con el sistema de oordenadas de mundo del ambiente. Esto quiere decir que siempre se inicia con la cámara irtual localizada en el origen de las coordenadas de mundo, apuntando hacia las oordenadas z positivas. Si necesitamos mover y/o apuntar la cámara a algún otro lugar, se eben definir transformaciones previas a la definición de la geometría del ambiente, para que fecten exclusivamente a la cámara. Esto se consigue declarando las rutinas de ransformación antes de la llamada a RiWorldBegin().

Aplicando un movimiento de traslación se le da una nueva posición a la cámara. Para irigirla hacia una dirección en particular, es suficiente especificar dos rotaciones, una sobre l eje x y otra en el eje y. La translación se debe aplicar primero porque la cámara se rotará obre la nueva posición donde será colocada. Sin embargo, las transformaciones deben eclararse en orden invertido, es decir que la primera debe ser la rotación sobre x, la egunda la rotación en y por último la translación. Esto es necesario debido al mecanismo de ila que usa *RenderMan* [28]. La translación se lleva a cabo con una llamada a función RiTranslate. Las rotaciones se determinan con un vector de dirección que indica hacia donde ebe apuntar la cámara. Los ángulos de rotación se calculan en la función ColocaZ. Para llo utiliza funciones trigonométricas básicas.

(a)

(b)

Fig. 4.3 Imágenes que muestran la utilización del modelo atmosférico propuesto. (a) puesta de sol, (b) medio ida despejado.

4.9 Lenguajes y herramientas de software.

Se utilizaron diferentes sistemas operativos, cada uno dependiente de la arquitectura de hardware:

- IRIX.
- Red Hat Linux.
- Windows 98
- Windows NT.

Se utilizó la versión 2.6 de *BMRT* como programa de "rendering" y como compilador de lenguaje de "shading".
Los compiladores de C++ fueron:

- CC de Silicon Graphics.
- GNU C++.
- Visual C++.

El modelador, para manipular la geometría fue *Rhinoceros* 1.0.
Para la creación de la animación se utilizó *Paint Shop Pro* 7.

4.10 Problemas en la creación del ambiente.

Se tuvieron algunas dificultades al momento de crear el prototipo. La selección del programa de "rendering" fue la principal. Inicialmente se intentó trabajar con *PhotoRealistic RenderMan*. En la Facultad de Ciencias se tiene instalada la versión 2.6, sin embargo esta no es una versión actualizada, por lo que algunas de las características que se requieren, no están soportadas. Por ejemplo los "shaders" de volumen no pueden contener sentencias de control illuminate, para el modelo atmosférico es importante contar con ella para considerar el color de la iluminación dentro de los cálculos que determinan color final del cielo. Esta fue una de las razones por las que se seleccionó *BMRT* como programa de "rendering". Sin embargo tampoco con esta herramienta estuvimos libres de problemas; que se relacionaron principalmente con las diferentes versiones de *BMRT*.

Mencionamos con anterioridad que en un archivo de *RenderMan* RIB o C, podemos incluir un archivo en formato RIB con un mecanismo similar a la directiva #include de C. Utilizamos este mecanismo para evitar convertir la descripción geométrica de la escena a código C, que llamamos la parte estática. Este mecanismo funciona perfectamente en la versión de *BMRT* para *Windows 98*. En *RedHat Linux*, cuando se generaban los cuadros y eran alimentados al programa de "rendering" en forma directa, solo era capaz de generar la imagen del primer cuadro, e interrumpía la ejecución del programa marcando un error. En *Windows NT* y en *IRIX*, no se reconocía el nombre de la función que permite la inclusión, como parte de la biblioteca, lo que nos llevo a pensar que tal vez no esta implementada en las versiones para esas plataforma. Esto generó un problema de compatibilidad serio, porque el mismo código no podía ejecutarse, sin modificaciones, en las diferentes máquinas donde se trabajo.

rremediablemente tuvimos que generar una versión compatible con *Windows* y otra para ambientes *IRIX* y *Linux*.

En *Windows* el código en C++ quedó compacto, porque el archivo en RIB proporcionaba la información de la geometría. Para solucionar el problema en *Windows NT*, se compiló el programa en *Windows 98*, y se copió el código ejecutable a NT, gracias a la compatibilidad del código ejecutable en ambas plataformas no hubo ningún problema.

Para el caso de *IRIX* y *Linux* se tuvo que traducir la geometría a C. El inconveniente de realizar esto, es que se genera un programa muy extenso, que requiere mucho tiempo y memoria para su compilación. Una PC con *Linux* con 64MB en RAM era incapaz de compilar el programa por limitaciones en la memoria RAM. Además la más pequeña modificación en alguna parte del programa, forzaba la compilación completa del código. Una solución para este problema podría ser seccionar el código en módulos para realizar compilación por separado. Nosotros optamos por compilar la geometría como en un solo modulo de compilación, en una computadora con suficiente memoria. La compilación genera el código objeto de la geometría que se puede integrar, esa parte estática, a la parte dinámica del modelo. Con esto se evita la compilación innecesaria de código cada vez que este se modifica.
La distribución de *BMRT* en *Windows* tiene un inconveniente con el entorno de trabajo en caso de que se use simultáneamente con el antivirus *Norton*. Si se tiene activado el compilador de "shader" no puede cumplir su tarea. Este problema esta documentado en la distribución misma de *BMRT* [14].

4.11 Tiempos de "rendering".

A continuación se muestran los tiempos que requiere la etapa de generación de imágenes en dos computadoras diferentes.

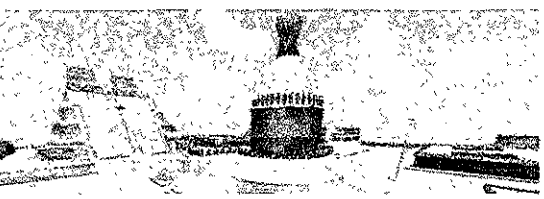
	Shader 1	Shader 2.
PC	3 minutos 30 segundos	16 minutos 30 segundos
Estación de trabajo.	1 minutos 34 segundos	14 minutos 00 segundos

La imagen de prueba que se genero fue la que se muestran en la fig. 4.5. Ambos son imágenes con una resolución de 600 X 400 pixeles.

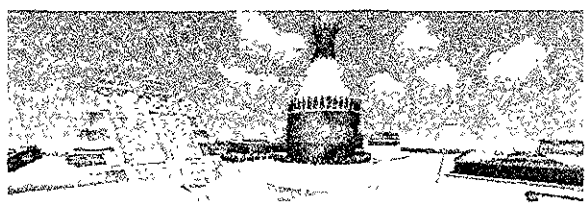
El shader1 es donde se programa el modelo atmosférico para escala intermedia. La resolución de la integral, en el calculo de la densidad se hace en forma analítica. El "shader" 2 es el mismo modelo para escala intermedia pero la resolución de la integral es por el método de cuadratura trapezoidal. La diferencia en el tiempo de procesamiento de imagen es notable.

as características de la PC son:
Procesador AMD K6-2 a 450 MHz.
160 MB en RAM.
8 MB en memoria de vídeo.
Sistema operativo Linux Red Hat 6.2.

a estación de trabajo es una 02 Silicon Graphics.
Procesador MIPS R10000 a 225 MHz.
128 MB en RAM,
Sistema operativo IRIX 6.5



(a)



(b)

ig. 4.6 Imágenes donde se muestra la diferencia al emplear (a) el "shader" donde la integral e la función de densidad se resuelve en forma analítica y (b) el "shader" donde se emplea el nétodo de cuadratura trapezoidal en la misma integral.

Conclusiones.

Sin duda uno de los elementos que incrementan el realismo en la generación de ambientes abiertos, es el modelo atmosférico.

El modelo atmosférico que se describió, emplea un algoritmo para aproximar el fenómeno de dispersión de Rayleigh. Este algoritmo divide la reacción que produce la atmósfera sobre la luz solar, en un vector de color. El vector es responsable de determinar el color del cielo y está directamente relacionado con la densidad atmosférica. Las variaciones en la densidad con la altitud, son aprovechadas en el modelo para asignar la coloración característica del cielo: el cenit de color azul y el horizonte de color blanco.

Este es un modelo simplificado, que resulta ser sencillo, intuitivo y útil; que toma en cuenta los fenómenos físicos que ocurren en la atmósfera terrestre. El modelo genera imágenes de gran calidad en poco tiempo, aun con computadoras personales, sin hardware especializado.

En el presente trabajo se expuso la teoría del fenómeno de dispersión, se revisaron las arquitecturas de dos sistemas para la generación de imágenes, y los programas de "rendering" que la soportan. Con esta información se diseño y programó, la simulación de un *ambiente virtual. En él se aplico el modelo para comprobar su eficacia.*

El resultado fue satisfactorio. Las imágenes se generaron con tal rapidez y calidad que fue posible crear una animación para mostrar todas las potencialidades del modelo.

Para enriquecer más al ambiente, se vio un método para generar nubes, que son fenómenos naturales de difícil modelado. Otros elementos que pueden ser considerados para futuros trabajos de investigación y que pueden ser iguales de complejos e interesantes, son la simulación de agua en movimiento, como en un río o lago; la simulación de fuego y el humo que produce, la lluvia, el viento entre otros. Cada uno de estos fenómenos naturales puede requerir de una técnica distinta, o una mezcla de varias de ellas. Muchas de las dificultades a las que nos enfrentamos al modelar estos fenómenos representan problemas abiertos y constantemente se proponen alternativas para su solución.

La principal contribución de este trabajo es la creación de un prototipo que es fácilmente aplicable a cualquier ambiente virtual. El diseño en su arquitectura, permite que cada elemento defina sus propiedades en forma independiente. Esto le proporciona mucha

xibilidad, misma que puede ser aprovechada al conjuntar estos elementos en la simulación un ambiente abierto. Se pueden generar imágenes y simulaciones con variadas sibilidades, teniendo la certeza de que se generarán en un lapso de tiempo razonable.

Anexo 1.

Variables globales definidas en el lenguaje de shading.

El lenguaje de "shading" define un conjunto de variables que proporcionan información de la geometría, sobre la que se lleva a cabo el proceso de "shading". Los valores que almacenan algunas de estas variables, son actualizados por el programa de "rendering" en forma directa, sin intervención del programador. Para otras, el valor se determina al ejecutarse las operaciones que se lleven a cabo en los "shaders". Los "shaders" pueden hacer uso de estas variables con solo nombrarlas, no es necesario definirlas porque son parte del lenguaje.
Para cada tipo de "shader" existe un conjunto de variables diferentes. A continuación listamos las variables globales para los tres tipos de "shaders" más importantes.

Shaders de superficie.

color	Cs	varying/uniform	Color de la superficie (entrada).
color	Os	varying/uniform	Opacidad de la superficie(entrada).
point	P	varying	Posición en la superficie.
point	dPdu	varying	Derivada de la posición en la superficie con respecto a u.
point	dPdv	varying	Derivada de la posición en la superficie con respecto a v.
point	N	varying	Normal a la superficie de "shading".
point	Ng	varying/uniform	Normal a la superficie geométrica.
float	u, v	varying	Parámetros de la superficie.
float	du, dv	varying/uniform	Cambio en los parámetros de la superficie.
float	s, t	varying	Coordenadas de la superficie de textura.
color	L	varying/uniform	Dirección de la fuente de luz a la superficie.
color	Cl	varying/uniform	Color de la luz.
point	I	varying	Dirección del rayo de luz que incide en el punto de la superficie (desde la cámara).
point	E	uniform	Posición de la cámara.
color	Ci	varying	Color de la luz desde la superficie (salida).
color	Oi	varying	Opacidad de la superficie (salida).

Shaders de fuente de luz.

point	P	varying	Posición de la fuente de luz.
point	dPdu	varying	Derivada de la posición con respecto a u.
point	dPdv	varying	Derivada de la posición con respecto a v.
point	N	varying	Normal a la superficie de "shading".
point	Ng	varying/uniform	Normal a la superficie geométrica.
float	u, v	varying	Parámetros de la superficie.
float	du, dv	varying/uniform	Cambio en los parámetros de la superficie.

float	s, t	varying	Coordenadas de la superficie de textura.
color	L	varying/uniform	Dirección de la fuente de luz a la superficie.
point	Ps	varying	Posición a ser iluminada.
point	E	uniform	Posición de la cámara.
float	ncomps	uniform	Número de componentes de color.
float	time	uniform	Tiempo actual de obturación.
color	Cl	varying	Color del rayo de luz.
color	Ol	varying	Opacidad del rayo de luz.

Shaders de volumen.

point	P	varying	Destino del rayo de luz.
point	E	uniform	Posición de la cámara.
point	I	varying	Dirección del rayo de luz.
color	Ci	varying	Color del rayo de luz en el destino.
color	Oi	varying	Opacidad del rayo de luz en el destino.
float	ncomps	uniform	Número de componentes de color.
float	time	uniform	Tiempo actual de obturación.
color	Ci	varying	Color del rayo de luz atenuado en el origen.
color	Oi	varying	Opacidad del rayo de luz atenuado en el origen.

Anexo 2.

Funciones integradas al lenguaje de "shading".

El lenguaje de "shading" tiene un conjunto de funciones preprogramadas listas para usarse. Muchas de estas funciones son idénticas a las que proporciona C en alguna de sus bibliotecas. A continuación se listan algunas de ellas, con un breve comentario en aquellas que no tengan contraparte en C o en las que su funcionalidad sea diferente.

Funciones matemáticas.
float radians(float d)
float degrees(float r)
float sin(float angle)
float cos(float angle)
float tan(float angle)
float asin(float f)
float acos(float f)
float atan(float y, x)
float atan(float y_over_x)
Los ángulos se expresan en radianes.

float pow(float x, float y)
float exp(float x)
float log(float x)
float log(float x, base)
float sqrt(float x)
float inversesqrt(float x)
Raíz cuadrada y 1/sqrt

float abs(float x)
float sign(float x)
float floor(float x)
float ceil(float x)
float round(float x)
float mod(float a, b)
Obtiene los mismos resultados que la función fmod de C, es decir que regresa a-b*floor(a/b)

float min(type a, b, ...)
float max(type a, b, ..)
float clamp(type x, minval, maxval)
Las funciones min y max regresa el valor mínimo y máximo respectivamente de entre una lista de valores. La función clamp regresa
min(max(x, minval), maxval)

esto es que clamp regresa minval si x es menor a minval, maxval si a es más grande que maxval, y a en cualquier otro caso.

ype mix(type x, y; float alpha)
La función mix regresa una combinación lineal, es decir
*(1-alpha) + y*alpha

loat step(float edge, x)
Regresa 0 si x es menor que edge y 1 si x es mayor o igual a edge.

loat smoothstep(float edge0, edge1, x)
Regresa 0 si x es menor o igual a edge0, y 1 si x es mayor o igual a edge1 y realiza una ntepolación de tipo Hermite entre 0 y 1 en el intervalo comprendido entre edge0 y edge1.

ype spline(type value; type v1, v2, ..., vn, vn1)
Ajusta un spline con interpolación Catmull-Rom para los puntos de control proporcionados v1, v2, ..., vn, vn1. Al menos se deben de dar 4 puntos de control. Si el valor de value es 0, la función regresa v2; si el valor de value es 1, regresa vn.

ype Du(type v)
ype Dv(type v)
ype Deriv(type num; float den)
Estas funciones calculan las derivadas de sus argumentos. Du y Dv calculan las derivadas en las direcciones u y v, respectivamente. Deriv calcula la derivada del primer argumento con respecto al segundo. Esto lo hace aplicando la regla de la cadena:
Deriv(num, den) = Du(num)/Du(den) + Dv(num)/Dv(den)

ype random()
Regresa un valor del tipo indicado cuyos componentes son números aleatorios entre 0 y 1.

Operaciones sobre color
loat comp(color c; float i)
Regresa el i-ésimo componente de color.
void setcomp(output color c; float i, float x)
Modifica el color c poniendo el valor x en el i-ésimo componente.
color ctransform(string tospacename; color c_rgb)
color ctransform(string fromspacename, tospacename; color c_from)
Transforma un color de una espacio de color a otro. La primera forma asume que c_rgb esta en el espacio "rgb".

Funciones geométricas.
float xcomp(ptype p)
float ycomp(ptype p)
float zcomp(ptype p)
float comp(ptype p; float i)
Regresan la coordenada x, y, z o al i-ésimo componente de la variable tipo punto.

void setxcomp(output ptype p; float x)
void setycomp(output ptype p; float x)
void setzcomp(output ptype p; float x)
void setcomp(output ptype p; float i, x)
Pone el valor a x, y, z o al i-ésimo componente de tipo punto.

float length(vector V)
float length(normal V)
Regresan la magnitud de un vector o una normal.
float distance(point P0, P1)
Regresa la distancia entre dos puntos.

Float ptlined(point P0, P1, Q)
Regresa la distancia desde Q al punto más cercano al segmento de línea que une a P0 y P1.

vector normalize(vector V)
vector normalize(normal V)
Regresan un vector en la misma dirección que V pero con magnitud de 1, esto es, V/length(V).

vector faceforward(vector N, I, Nref)
vector faceforward(vector N, I)
Si Nref.I < 0, regresa N; de lo contrario, regresa -N. Para la versión de dos argumentos, Nref *corresponde a Ng, la superficie normal del objeto. El objetivo de esta rutina* es regresar una versión de N que su dirección sea hacia la cámara.

vector reflect(vector I, N)
Para un vector incidente I y una normal de "shading" N, regresa la dirección de reflexión R = I-2*(N.I)*N.

vector refract(vector I, N; float eta)
Para un vector incidente I y una normal de "shading" N, regresa la dirección de refracción empleando la ley de Snell. El parámetro eta es el radio del índice de refracción del volumen que contiene I dividido por el índice de refracción del volumen.

point transform(string tospacename; point p_current)
vector vtransform(string tospacename; vector v_current)
normal vtransform(string tospacename; normal n_current)
Transforma un punto, vector o normal a un sistema de coordenadas tospacename.

point transform(string fromspacename, tospacename; point pfrom)
vector vtransform(string fromspacename, tospacename; vector vfrom)
normal ntransform(string fromspacename, tospacename; normal nfrom)
Transforman un punto, vector o normal del sistema de coordenadas fromspacename al sistema de coordenadas tospacename.

point tranform(matrix tospace; point p_current)

ector vtranform(matrix tospace; vector v_current)
ormal ntranform(matrix tospace; normal n_current)

oint transform(string fromspacename; matrix tospace; point pfrom)
ector vtransform(string fromspacename; matrix tospace; vector vfrom)
ormal ntransform(string fromspacename; matrix tospace; normal nfrom)
Estas rutinas funciones exactamente como las que usan nombres de espacios pero utilizan en su lugar matrices de transformación

oint rotate(point Q; float angle; point P0, P1)
Regresa el punto calculado al rotar el punto Q un ángulo angle en radianes alrededor del eje que pasa desde el punto P0 al punto P1.

Funciones de matrices.
oat determinant(matrix m)
Regresa el determinante de la matrix m.

natrix translate(matrix m; point t)
natrix rotate(matrix m; float angle; vector axis)
natrix scale(uniform matrix m; uniform point t)
Regresa una matriz que es el resultado de agregar transformaciones simples en la matriz m. En cada caso, todos los argumentos de estas funciones deben ser uniformes.

Anexo 3.

"Shaders" creados para la simulación de ambientes abiertos.

El siguiente es el "shader" de volumen que implementa el modelo atmosférico.

```
/*****************************************************************************
 Universidad Nacional Autonoma de Mexico.
 Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
 Shader del modelo atmosferico.
 Programado en lenguaje de shading. BMRT 2.6
 Miguel Miranda Miranda.
 *****************************************************************************/

volume cielo (float densidad = 60, decremento = 400;
              float k = 50;
              float iluminacion = 1;
              color colorilum = 1; )
{
  float tau;
  point origen = transform ("shader", P-I);
  vector incidente = vtransform ("shader", I);
  vector IN = normalize (incidente);
  color Cv = 0, Ov = 0;           /* color y opacidad del volumen */
  float te, z0, zd;
  color li, luztau;
  point PP;

  te = 3.0;
  /* Asigna el punto inicial de integracion */
  z0 = zcomp(origen);
  zd = zcomp(IN);

  li = 0;
  if ( abs(zd) > 0.000000001)
     tau = abs( (densidad*te) * ( exp(-(z0+zd*te)) - exp(-z0) ) / (decremento/zd)
);
  else
     tau = abs(densidad * te * exp( -(decremento*z0) ) );

  if (iluminacion > 0) {
     point PW = transform ("shader", "current", PP);
     illuminance (PW) { li += Cl; }
  } else { li = colorilum; }

Ov = color( exp(-tau*21), exp(-tau*2.25), exp(-tau) );
  Cv = li*Ov;

  Ci = Cv + (1-Ov)*Ci;
  Oi = Ov + (1-Ov)*Oi;
```

```
*  printf(" Ci: %c Oi: %c\n", Ci, Oi);*/
```

A continuación se muestra el "shader" que declara el mismo modelo, pero utilizando un método numérico para calcular la distribución de densidad atmosférica.

```
*************************************************************************

*************************************************************************/

/*
*Funcion de densidad atmosferica.
*/
#define DensidadAtmosferica(PP,densidad,decremento,iluminacion,g,li,colorilum) \
\
    li = 0;\
    g = (densidad * exp(-decremento*zcomp(PP)));\
    if (iluminacion > 0) {\
      point PW = transform ("shader", "current", PP);\
      illuminance (PW) { li += Cl; }\
    } else { li = colorilum; }\

volume
cielo (float densidad = 60, decremento = 400;
           float iniciointegral = 0, finintegral = 100;
           float incrementomin = 0.01, incrementomax = 2.5;
           float k = 50;
           float debug = 0;
           float iluminacion = 1;
color colorilum = 1;
    )
{
  float t, tau;
  point origen = transform ("shader", P-I);
  vector incidente = vtransform ("shader", I);
  vector IN = normalize (incidente);
  color Cv = 0, Ov = 0;           /* color y opacidad del volumen */
  color dC, dO;                   /* diferencial de color y opacidad */
  float ss, dtau, ult_dtau, tc, te;
  float nincrementos = 0;          /* record number of integration steps */
  color li, ult_li, luztau;
  point PP;

  /*
   * Calcula el punto final de integracion.
   */
  te = min (length (incidente), finintegral) - 0.0001;
  /* Asigna el punto inicial de integracion */
  tc = iniciointegral;

  /*
   *Proceso de integracion
   */
  t = tc;
  if (t < te) {
```

```
      PP = origen + t * IN;
      DensidadAtmosferica (PP, densidad, decremento, iluminacion, dtau, li,
colorilum);
      ss = random() * min (clamp (1/(k*dtau+.001), incrementomin, incrementomax),
te-t);
      t += ss;
      nincrementos += 1;

      while (t <= te) {
        ult_dtau = dtau;
        ult_li = li;
        PP = origen + t*IN;
        DensidadAtmosferica (PP, densidad, decremento, iluminacion, dtau, li,
colorilum);

        /*
         * Calculamos dC y dO, el color y la opacidad de la porcion
         * del volumen considerado hasta este momento
         */
        tau = .5 * ss * (dtau + ult_dtau);
        luztau = .5 * ss * (li*dtau + ult_li*ult_dtau);

        dO = 1 - color (exp(-tau), exp(-tau*2.25), exp(-tau*21));
        dC = luztau * dO;

        /*
r        * Ahora calculamos Cv/Ov tomando en cuenta dC y dO
         */
        Cv += (1-Ov)*dC;
        Ov += (1-Ov)*dO;

        ss = max (min (clamp (1/(k*dtau+.001), incrementomin, incrementomax), te-
t), 0.0005);
        t += ss;
        nincrementos += 1;
/*      printf (" t = %f (te = %f, ss = %f)\n", t, te, ss); */
        }
    }

  /* Ci y Oi son el color (premultiplicado por la opacidad) y la opacidad
   * del elemento de fondo (en este caso la esfera).
   * Ahora Cv es la contribucion de luz del volumen, y Ov es la opacidad
   * del volumen. (1-Ov)*Ci es la luz desde el fondo que es modificada
   * por el volumen.
   */
  Ci = 15*Cv + (1-Ov)*Ci;
  Oi = Ov + (1-Ov)*Oi;

  if (debug > 0) {
      printf ("nincrementos = %f, tc = %f, t1 = %f, te = %f\n", nincrementos, tc,
iniciointegral, te);
      printf ("   Cv = %c, Ov = %c\n", Cv, Ov);
    }
}
```

l siguiente es el "shader" de fuente de luz que simula la iluminación directa proveniente del ol.

```
/*****************************************************************************
Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Shader para simular la iluminacion del sol, considerando su posicion
en el cielo.
Programado en lenguaje de shading. BMRT 2.6
Miguel Miranda Miranda.
*****************************************************************************/

ight luzdia(float intensity = 1;
            color lightcolor = 1;
            point sol = (0, 0, -1);)

  point D = transform("world", sol);
  solar (D, PI)
     Cl = intensity*lightcolor;
```

Por último tenemos el "shader" de superficie que crea la apariencia de las nubes.

```
/*****************************************************************************
Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Shader para la generacion de nubes.
Programado en lenguaje de shading. BMRT 2.6
Miguel Miranda Miranda.
*****************************************************************************/

/* Funcion noise() con un rango entre [-1 1] */
#define snoise(p) (2 * (float noise(p)) - 1)

/* Funcion noise() filtrada con una funcion cuadrada suavizada */
#define snoisefiltrado(p,ancho) (snoise(p) * (1-smoothstep (0.2,0.75,ancho)))

/* Funcion fractional Browmanian noise() que forma el patron irregular
     dentro de la nube */
float fBm (point p, movimiento; float anchofiltro;
           uniform float octavas, lagunaridad, ganancia)

    uniform float amp = 1;
    varying point pp = p;
    varying float sum = 0, fw = anchofiltro;
    uniform float i;

    for (i = 0;  i < octavas;  i += 1) {
      sum += amp * snoisefiltrado (pp+movimiento, fw);
      amp *= ganancia;  pp *= lagunaridad;  fw *= lagunaridad;
    }
    return sum;
}

/* Calcula la densidad del volumen en una muestra */
```

```
:loat densidadvolumen (point Pobj, movimiento; float freqnoise, incremento)

    float densidad = 0.5 + 0.5 * fBm(Pobj*freqnoise, movimiento,
ncremento*freqnoise,
                                7, 2, 0.65);
    /* Incrementa el Contraste */
    densidad = pow(clamp(densidad,0,1), 7.28);
    return densidad;
}

/* Proporciona el color del volumen */
color colorvolumen (point Pobj)
{
    return color 1;  /* Nubes blancas */
}

/* Calcula las sombras en las muestras */
float sombravolumen (point Pobj, movimiento; vector Lobj;
                     float densidad, freqnoise, incremento, fin)
{
    float Oi = 0;
    float Llen = length(Lobj);
    vector Iobj = normalize(Lobj);
    float t0, t1;

    float final = fin;  /* distance to march */
    fin = min (final, Llen);
    float d = 0;
    float ss = min (incremento, final-d);
    float ult_dtau = densidadvolumen (Pobj, movimiento, freqnoise, incremento);
    while (d <= final) {
      /* Incrementa. Calcula la luz y la densidad */
      ss = clamp (ss, 0.005, fin-d);
      d += ss;
      float dtau = densidadvolumen (Pobj + d*Iobj, movimiento, freqnoise,
incremento);
      float tau = densidad * ss/2 * (dtau + ult_dtau);
      Oi += (1-Oi) * (1 - exp(-tau));
      ult_dtau = dtau;
    }
    return Oi;
}

/* Calcula el color de la luz en las muestras */
color luzvolumen (point Pactual, Pobj, movimiento;
                  float densidad, freqnoise, incremento, fin)
{
    color Ldispersion = 0;
    illuminance (Pactual) {
      extern color Cl;
      extern vector L;
      color Cdispersion = Cl;
      if (densidad > 0)
          Cdispersion *= 1 - sombravolumen (Pobj, movimiento,
vtransform("object",L), densidad, freqnoise, incremento, fin);
      Ldispersion += Cdispersion;
    }
```

```
    return Ldispersion * colorvolumen(Pobj);
}


/*****************************************************************
 Declaracion del shader
*****************************************************************/
surface nubes (float Ka = 0.127, Kd = 1;
            float densidadopacidad = 1, densidadluz = 1, densidadsombra = 1;
            float incremento = 0.1, incrementosombra = 0.5;
            float freqnoise = 2.0;
            point movimiento = 0.1; )
{
    Oi = 0;
    /* Realiza el shading solo en la parte posterior de la superficie */
    if (N.I > 0) {
      /* Encuentra el segmento de rayo que recorrera. El punto final mas
         * lejano es P. Los otros puntos se encuentran por ray tracing
       * contra la esfera (en direccion opuesta). */
      point  Pobj = transform ("object", P);
      vector Iobj = normalize (vtransform ("object", -I));
      float t0 = 0;

      float fin = 20;  /* distancia a recorrer */

      point origen = Pobj - t0*Iobj;

      point Worigen = transform ("object", "current", origen);

      /* Integra hacia adelante desde el punto inicial */
      float d = random()*incremento;

      /* Calcula un incremento razonable */
      float ss = min (incremento, fin-d);

      point Psamp = origen + d*Iobj;
      float ult_dtau = densidadvolumen (Psamp, movimiento, freqnoise, incremento);
      color ult_li = luzvolumen (transform ("object", "current", Psamp),
                           Psamp, movimiento, densidadsombra,
                           freqnoise, incrementosombra, fin);
      while (d <= fin) {
          /* Da un incremento y obtiene la dispersion de la luz y densidad */
          ss = clamp (ss, 0.005, fin-d);
          d += ss;
          /* Calcula la dispersion de luz y densidad */
          Psamp = origen + d*Iobj;
          float dtau = densidadvolumen (Psamp, movimiento, freqnoise, incremento);
          color li = luzvolumen (transform ("object", "current", Psamp),
                           Psamp, movimiento, densidadsombra,
                           freqnoise, incrementosombra, fin);

          float tau = densidadopacidad * ss/2 * (dtau + ult_dtau);
          color luztau = densidadluz * ss/2 * (li*dtau + ult_li*ult_dtau);

          /* Composicion de la luz de fondo con extincion exponencial */
          Ci += (1-Oi) * luztau;
          Oi += (1-Oi) * (1 - exp(-tau));
```

Anexo 4.

Programa traductor de formato RIB a C.

El siguiente programa lee un archivo en *RenderMan Interface ByteStream Protocol* y produce un programa con el conjunto de llamadas de la API de *RenderMan* para usarse en lenguaje C. Su finalidad es proporcionar el código que generará la animación de la escena. El código en RIB fue generado con un modelador.

```
/*****************************************************************
Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas

Programa convertidor de archivos para RenderMan. De RIB a C con
el proposito de poder programar animaciones completas.
Archivo Rib2C.h
Programo: Miguel Miranda Miranda. 8718694-5.
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <stream.h>
#include <string.h>

#define DIGITOS 5   /* Define hasta cuantos digitos puede procesar
                    la funcion Entero2Alpha */
class Archivo {
  FILE* fp;
  FILE* archsal;
  char c;
  unsigned int pointpoligonnum;
  void ObtenToken();
  void Comentario();
  void Token();
  int Arreglo(char*, char*);
  int BuscaProcedimientoComun(char*);
  int BuscaProcedimientoComunNULL(char*);
  int BuscaPointsPolygons(char*);
  int BuscaColor(char*);
  void Color(char*);

  void ProcedimientoComun(char*);
  void ProcedimientoComunNULL(char*);
  void PointsPolygons(char*);
  void Entero2Alpha(unsigned int);
  char numerocar[DIGITOS];

 public:
  Archivo();
  void Abre_Archivo(char*);
```

```
 void EscribeArchivoC(char*);
;

*****************************************************************
Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas aplicadas y sistemas

Programa convertidor de archivos para RenderMan. De RIB a C con
el proposito de poder programar animaciones completas.
Archivo Rib2C.C
Programo: Miguel Miranda Miranda. 8718694-5.
*****************************************************************/

#include "rib2c.h"

Archivo::Archivo( ) {
  pointpoligonnum = 0;  /* Contador de numero de poligonos */


void Archivo::Abre_Archivo(char nomarch[40]) {
   if ((fp = fopen(nomarch, "r")) == NULL)
      { printf("\nError. No se puede abrir el archivo\n");
      exit(1);  }
}


void Archivo::EscribeArchivoC(char nomarch[40]) {

   archsal = fopen(nomarch, "wb");
   if ( !archsal ) {
      cout << "No pude abrir el archivo C para escribir";
      return;
   }

   ObtenToken();
}


void Archivo::ObtenToken() {
   while ( c != EOF ) {
      c = getc(fp);
      if (c == '#')              /* Comentarios */
      Comentario();

      if ( (c >= 65) && (c <= 90) || (c >= 97) && (c <= 122) ) /* Tokens */
                  Token();   /* Token */
   }
}

/* Provisionalmente los comentarios los ignora */
void Archivo::Comentario() {
   while (c != '\n')
      c = getc(fp);
}

void Archivo::Token() {
```

```
  int i = 0; /* Contador de caracteres del token */
  char token[25];
  int identifica;  /* Identifica de que tipo de token tratamos */

  while ( (c >= 65) && (c <= 90) || (c >= 97) && (c <= 122) ) {
     /*   while ( (c != ' ') && (c != '\n') && (c != '[')  )  { */
      token[i] = c;
      i++;
      c = getc(fp);
  }
  token[i] = '\0';  /* Fin del token */

  if ( !BuscaPointsPolygons(token) )
     PointsPolygons(token);
  else
     if ( !BuscaProcedimientoComunNULL(token) )
        ProcedimientoComunNULL(token);
     else
        if ( !BuscaProcedimientoComun(token) )
           ProcedimientoComun(token);
        else
           if ( !BuscaColor(token) )
           Color(token);
}

/* Procedimiento comun se refiere a todos los procedimientos en RenderMan */
/* que requieren solamente agregar los parentesis y las comas a los */
/* parametros del procedimiento */

void Archivo::ProcedimientoComun(char* token) {
   fputc('R', archsal);
   fputc('i', archsal);
   fwrite( (void*) token, strlen(token), 1, archsal);
   fputc('(', archsal);
   while (c != '\n') {
      while (c == ' ')
         c = getc(fp);
      while ( (c != ' ') && (c != '\n') ) {
         fputc( c, archsal);
         c = getc(fp);
      }
      while (c == ' ')
         c = getc(fp);
      if ( c != '\n' )
         fputc(',', archsal);
   }

   fputc(')', archsal);
   fputc(';', archsal);
   fputc('\n', archsal);
}


/* Procedimiento comun NULL se refiere a todos los procedimientos en */
/* RenderMan que requieren agregar parentesis, comas y la cadena RI_NULL */
/* a los parametros del procedimiento */
```

```
void Archivo::ProcedimientoComunNULL(char* token) {
   char cadRI_NULL[8];

   strcpy(cadRI_NULL, "RI_NULL\0");
   fputc('R', archsal);
   fputc('i', archsal);
   fwrite( (void*) token, strlen(token), 1, archsal);
   fputc('(', archsal);
   while (c != '\n') {
      while (c == ' ')
         c = getc(fp);
      while ( (c != ' ') && (c != '\n') ) {
         fputc( c, archsal);
         c = getc(fp);
      }
      while (c == ' ')
         c = getc(fp);
      fputc(',', archsal);
   }

   fwrite( (void*)cadRI_NULL, strlen(cadRI_NULL), 1, archsal);
   fputc(')', archsal);
   fputc(';', archsal);
   fputc('\n', archsal);
}


/* Procedimiento Poligonos se refiere a los procedimientos en RenderMan */
/* en donde se tiene que cambiar al formato de C */

int Archivo::Arreglo(char* nombre, char* cadtipo) {
   char nomarreglo[15];   /* Guarda el nombre del arreglo de Nvertices */
   char numero[15];         /* Guarda el numero actual en alfanumerico */
   int i;
   unsigned int npol = 0;   /* Numero total de poligonos */
   int bandera = 0;         /* Bandera que determina si se capturo numero */

   fwrite( (void*)cadtipo, strlen(cadtipo), 1, archsal);
   fputc(' ', archsal);
   fwrite( (void*)nombre, strlen(nombre), 1, archsal);
   fputc('[', archsal);
   fputc(']', archsal);
   fputc('=', archsal);
   fputc('{', archsal);

   while (c == ' ')
       c = getc(fp);

   while ( (c != ']') ) {
     i = 0;
     bandera = 0;
     /*       while ( (c == ' ') || (c == '[') || (c == '\n') || (c == '\t') )
           c = getc(fp);*/
      /*      while ( (c != ' ') && (c != ']') && (c != '\n') ) { */
```

```
    while ( ( (c >= '0') && (c <= '9') ) || (c == '-') || (c == '.') || (c ==
e') ) {
        bandera = 1;
        numero[i] = c;
        i++;
        c = getc(fp);
     }
    if (bandera) {
        numero[i] = '\0';  /* Fin del token */
        npol++;
        fwrite( (void*) numero, strlen(numero), 1, archsal);
        fputc(',', archsal);
        fputc(' ', archsal);
    }

    /*      while ( (c == ' ') || (c == '\n') || (c == '\t') )
          c = getc(fp);*/

    /*      fwrite( (void*) numero, strlen(numero), 1, archsal); */
    /*      if ( (c != ']') && (c != '\n') ) {
        fputc(',', archsal);
        fputc(' ', archsal);
      }*/
    c = getc(fp);
  }
  c = getc(fp);
  fputc('}', archsal);
  fputc(';', archsal);
  fputc('\n', archsal);
  return npol;
}

void Archivo::PointsPolygons(char* token) {

  char cadRI_NULL[8];       /* Guarda la cadena RI_NULL */
  char cadRI_P[5];          /* Guarda la cadena RI_P */
  char cadRI_N[5];          /* Guarda la cadena RI_N */
  char cadRtPointer[12];    /* Guarda la cadena RtPointer */
  char cadnumerover[15];    /* Guarda el nombre del arreglo de Nvertices */
  char cadvertices[15];     /* Guarda el nombre del arreglo de vertices */
  char cadpuntos[15];       /* Guarda el nombre del arreglo de puntos */
  char cadnormales[15];     /* Guarda el nombre del arreglo de normales */
  char cadnumpoligonos[5]; /* Guarda el numero de Polygons en el PointPol */
  char cadint[4];           /* Guarda la cadena int */
  char cadRtPoint[8];       /* Guarda la cadena RtPoint */
  unsigned int npol, basura;


  pointpoligonnum++;     /* Es el numero de PointPoligon. */
  strcpy(cadRtPointer, "(RtPointer)\0" );
  strcpy(cadRI_NULL, "RI_NULL\0");
  strcpy(cadRI_P, "RI_P\0");
  strcpy(cadRI_N, "RI_N\0");
  strcpy(cadint, "int\0");
  strcpy(cadRtPoint, "RtPoint\0");

  /**************************************************************/
```

```
/* Construye el arreglo del Numero de vertices                    */

strcpy(cadnumerover, "NVerts\0");
Entero2Alpha(pointpoligonnum);
strcat(cadnumerover, numerocar);
npol = Arreglo(cadnumerover, cadint);

/*****************************************************************/
/* Construye el arreglo de los vertices                           */

strcpy(cadvertices, "Verts\0");
strcat(cadvertices, numerocar);
basura = Arreglo(cadvertices, cadint);

/*****************************************************************/
/* Construye el arreglo de puntos                                 */

strcpy(cadpuntos, "Puntos\0");
strcat(cadpuntos, numerocar);
basura = Arreglo(cadpuntos, cadRtPoint);

/*****************************************************************/
/* Construye el arreglo de normales                               */

strcpy(cadnormales, "Normal\0");
strcat(cadnormales, numerocar);
basura = Arreglo(cadnormales, cadRtPoint);

/*****************************************************************/
/* Construye la llamada al procedimiento PointPolygon             */

fputc('R', archsal);
fputc('i', archsal);
fwrite( (void*) token, strlen(token), 1, archsal);
fputc('(', archsal);

Entero2Alpha(npol);
strcpy (cadnumpoligonos, numerocar);
fwrite( (void*) cadnumpoligonos, strlen(cadnumpoligonos), 1, archsal);
fputc(',', archsal);
fputc(' ', archsal);
fwrite( (void*) cadnumerover, strlen(cadnumerover), 1, archsal);
fputc(',', archsal);
fputc(' ', archsal);
fwrite( (void*) cadvertices, strlen(cadvertices), 1, archsal);
fputc(',', archsal);
fputc(' ', archsal);
fwrite( (void*) cadRI_P, strlen(cadRI_P), 1, archsal);
fputc(',', archsal);
fputc(' ', archsal);
fwrite( (void*) cadRtPointer, strlen(cadRtPointer), 1, archsal);
fwrite( (void*) cadpuntos, strlen(cadpuntos), 1, archsal);
fputc(',', archsal);
fputc(' ', archsal);
fwrite( (void*) cadRI_N, strlen(cadRI_N), 1, archsal);
fputc(',', archsal);
fputc(' ', archsal);
```

```
p    fwrite( (void*) cadRtPointer, strlen(cadRtPointer), 1, archsal);
   fwrite( (void*) cadnormales, strlen(cadnormales), 1, archsal);
   fputc(',', archsal);
   fputc(' ', archsal);
   fwrite( (void*) cadRI_NULL, strlen(cadRI_NULL), 1, archsal);
   fputc(')', archsal);
   fputc(';', archsal);
   fputc('\n', archsal);



nt Archivo::BuscaProcedimientoComun(char* cadena) {
   char *tabla[] = { "Declare", "Translate", "Rotate", "FrameBegin",
          "FrameEnd", "WorldBegin", "WorldEnd", "End",
          "AttributeBegin", "AttributeEnd", "Format", "Transform", "Bound",
          "Clipping", "ColorSamples", "ConcatTransform", "CoordinateSystem",
          "CropWindow", "DepthOfField", "Detail", "DetailRange",
          "ErrorHandler", "Exposure", "FrameAspectRatio",
          "GeometricAproximation", "Identity",
          "Illuminate", "RiMatte", "MotionBegin", "MotionEnd", "ObjectBegin",
          "ObjectEnd", "ObjectInstance", "Opacity", "Orientation",
          "Perspective", "PixelFilter", "PixelSamples", "PixelVariance",
          "Procedural", "RelativeDetail", "ReverseOrientation",
          "ScreenWindow", "ShadingRate", "Shutter", "Sides", "SolidBegin",
          "SolidEnd", "TextureCoordinates", "TransformPoints",
                "TrimCurve" };

   int i;
   int elementosenarreglo = 51;
  for (i = 0; i < elementosenarreglo; i++) {
       if ( strcmp(cadena, tabla[i] ) ==  0 )
            return 0;
   }
   return 1;
}

int Archivo::BuscaProcedimientoComunNULL(char* cadena) {
   char *tabla[] = { "Surface", "LightSource", "Begin", "Display",
          "Projection", "Cone",
          "Disk", "Sphere", "Torus", "Cylinder", "Hiperboloid", "Paraboloid",
          "AreaLightSource", "Atmosphere", "Attribute", "Deformation",
          "Displacement", "Exterior", "Hider", "Imager", "Interior",
          "MakeBump", "MakeCubeFaceEnvironment", "MakeLatLongEnvironment",
                "MakeShadow", "MakeTexture", "Option" };

   int i;
   int elementosenarreglo = 27;
   for (i = 0; i < elementosenarreglo; i++) {
       if ( strcmp(cadena, tabla[i]) == 0 )
            return 0;
   }
   return 1;
}

int Archivo::BuscaPointsPolygons(char* cadena) {
  char *tabla[] = { "PointsPolygons" };
```

```
   int i;
   int elementosenarreglo = 1;
   for (i = 0; i < elementosenarreglo; i++) {
        if ( strcmp(cadena, tabla[i]) == 0 )
             return 0;
   }
   return 1;
}

int Archivo::BuscaColor(char* cadena) {
  char *tabla[] = { "Color" };

   int i;
   int elementosenarreglo = 1;
   for (i = 0; i < elementosenarreglo; i++) {
        if ( strcmp(cadena, tabla[i]) == 0 )
             return 0;
   }
   return 1;

}

void Archivo::Color(char* token) {
   char cadcolor[9];
   char cadRiColor[17];

   strcpy(cadcolor, "color[0]\0");
   fwrite( (void*) cadcolor, strlen(cadcolor), 1, archsal);
   fputc('=', archsal);

   while ( (c == ' ') || (c == '[') )
      c = getc(fp);
   while ( (c != ' ') && (c != '\n') ) {
      fputc( c, archsal);
      c = getc(fp);
   }
   fputc(';', archsal);
   fputc('\n', archsal);

   strcpy(cadcolor, "color[1]\0");
   fwrite( (void*) cadcolor, strlen(cadcolor), 1, archsal);
   fputc('=', archsal);

   while (c == ' ')
      c = getc(fp);
   while (c != ' ') {
      fputc( c, archsal);
      c = getc(fp);
   }
   fputc(';', archsal);
   fputc('\n', archsal);

   strcpy(cadcolor, "color[2]\0");
   fwrite( (void*) cadcolor, strlen(cadcolor), 1, archsal);
   fputc('=', archsal);

   while (c == ' ')
```

```
    c = getc(fp);
  while (c != ' ') {
    fputc( c, archsal);
    c = getc(fp);
  }
  fputc(';', archsal);
  fputc('\n', archsal);

  strcpy(cadRiColor, "RiColor(color);\n\0'");
  fwrite( (void*) cadRiColor, strlen(cadRiColor), 1, archsal);


void Archivo::Entero2Alpha(unsigned int n) {
  int i, j;              /* Iterador */
  unsigned short bandera = 0;  /* Bandera para empezar a escribir numeros */
                               /* Si es cero no escribe 0's que no son */
                               /* significativos por ejemplo 00047 */
  unsigned int division;       /* Contiene la division que determina */
                               /* el numero que se colocara en numero */
  int p = 0;          /* Apuntador en la cadena numero */
  int potencia;

  /*   if (n > (pow(10, DIGITOS)-1) )
     exit(1);*/

  for (i = DIGITOS-1; i >= 0; i--) {

    /* Eleva a una potencia de 10^i */
    potencia = 1;
    for (j = i; j > 0; j--)
      potencia *= 10 ;

    division = n / potencia;
    n = n - (division * potencia );
    /*
    printf( " %i %i %i", potencia, division, n);
    */
    if (division != 0)
        bandera = 1;
    if (bandera != 0) {
        switch(division) {
           case 0 : numerocar[p] = '0';
                    break;
           case 1 : numerocar[p] = '1';
                    break;
           case 2 : numerocar[p] = '2';
                    break;
           case 3 : numerocar[p] = '3';
                    break;
           case 4 : numerocar[p] = '4';
                    break;
           case 5 : numerocar[p] = '5';
                    break;
           case 6 : numerocar[p] = '6';
                    break;
           case 7 : numerocar[p] = '7';
```

```
                    break;
            case 8 : numerocar[p] = '8';
                    break;
            case 9 : numerocar[p] = '9';
                    break;
        }
        p++;
      }
    }
    numerocar[p] = '\0';
}


main() {
    Archivo arch;

    arch.Abre_Archivo("mmm.rib");
    arch.EscribeArchivoC("mmm.C");
}
```

Anexo 5.

Programas en C++.

A continuación se lista el archivo de encabezado que define la clase Camara. Posteriormente se muestra el archivo de código que declara esta clase. Los objetos de esta clase son responsables de colocar la cámara en la posición definida por el usuario, y dirigir la vista de esta cámara en la dirección que se le indica.

```
/*****************************************************************
 Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Modelo de camara.
Definicion de la Clase Camara
Archivo camara.h
Miguel Miranda Miranda.
*****************************************************************/

#include <ri.h>
#include <math.h>

#ifndef __CAMARA_H
#define __CAMARA_H

#ifndef PI
#define PI 3.14159265359
#endif

class Camara {
  float rotx, roty;
  RtPoint posicion, direccion;
  void ColocaZ();
  void ColocaCamara();
 public:
  Camara(RtPoint, RtPoint);
  void ColocaCamara(RtPoint, RtPoint);
  float DameRotacionX();
  float DameRotacionY();
  float* DamePosicion();
  float* DameDireccion();
};

#endif

/*****************************************************************
 Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Modelo de camara. Se asume que la geometria esta sobre el plano
xy.
Archivo camara.C
Miguel Miranda Miranda.
*****************************************************************/
```

```
#include "camara.h"

Camara::Camara(RtPoint pos, RtPoint dir) {
  posicion[0] = pos[0];
  posicion[1] = pos[1];
  posicion[2] = pos[2];
  direccion[0] = dir[0];
  direccion[1] = dir[1];
  direccion[2] = dir[2];

  ColocaCamara();
}

/*************************************************************************
 Funcion que realiza las rotaciones de la camara para colocar la
 direccion del eje de las z's
*************************************************************************/

void Camara::ColocaZ() {
  double proyxz, proyyz;

  if (direccion[0] == 0 && direccion[1] == 0 && direccion[2] == 0)
    return;

  proyxz = sqrt(direccion[0]*direccion[0] + direccion[2]*direccion[2]);
  if (proyxz == 0)
    roty = (direccion[1] < 0) ? 180 : 0;
  else
    roty = 180 * acos( direccion[2]/proyxz) / PI;

  proyyz = sqrt(direccion[1]*direccion[1] + proyxz*proyxz);
  rotx = 180 * acos( proyxz/proyyz) / PI;

  if (direccion[1] <= 0)
    rotx = -rotx;
   RiRotate(rotx, 1.0, 0.0, 0.0);

  if (direccion[0] > 0)
    roty = -roty;
  RiRotate(roty, 0.0, 1.0, 0.0);
}

/*************************************************************************
  Funcion que declara el modelo de camara. Primero se rota 90 grados
  sobre x, despues se traslada la camara y por ultimo se invoca a la
  funcion que direcciona la camara
*************************************************************************/

void Camara::ColocaCamara(){
  RiIdentity();
  ColocaZ();
  RiTranslate(-posicion[0], -posicion[1], -posicion[2]);
  RiRotate(90, -1.0, 0.0, 0.0);
}

/*************************************************************************
```

```
  Funcion que declara el modelo de camara. Primero se rota 90 grados
  sobre x, despues se traslada la camara y por ultimo se invoca a la
  funcion que direcciona la camara
*********************************************************************/

void Camara::ColocaCamara(RtPoint pos, RtPoint dir){
  posicion[0] = pos[0];
  posicion[1] = pos[1];
  posicion[2] = pos[2];
  direccion[0] = dir[0];
  direccion[1] = dir[1];
  direccion[2] = dir[2];

  RiIdentity();
  ColocaZ();
  RiTranslate(-posicion[0], -posicion[1], -posicion[2]);
  RiRotate(90, -1.0, 0.0, 0.0);
}

float Camara::DameRotacionX() {
  return rotx;
}

float Camara::DameRotacionY() {
  return roty;
}

float* Camara::DamePosicion() {
  return posicion;
}

float* Camara::DameDireccion() {
  return direccion;
}
```

En seguida se muestra el archivo de encabezado que define la clase BovedaCeleste. Después se lista el archivo de código que declara la clase. Los objetos de esta clase son responsables de crear una semiesfera; definir los parámetros necesarios que requiere el "shader" que implementa el modelo atmosférico y realizar las transformaciones necesarias para que el "shader" se evalúe en forma correcta.

```
/*****************************************************************
 Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Modelo atmosferico.
Declaracion de la clase BovedaCeleste
Archivo bovedaceleste.h
Miguel Miranda Miranda.
*****************************************************************/


#include <math.h>
#include <ri.h>

#ifndef __BOVEDACELESTE_H
```

```
#define __BOVEDACELESTE_H

#ifndef PI
#define PI 3.14159265359
#endif

class BovedaCeleste {
  RtString casts_shadows;
  RtFloat iniciointegral;
  RtFloat finintegral;
  RtFloat k;
  RtFloat incrementomax;
  RtFloat incrementomin;
  RtFloat decremento;
  RtFloat densidad;
  RtFloat iluminacion;
  RtFloat debug;
  RtFloat radio;
  RtColor color;
  RtPoint posicion, direccion;
  double proyxz;
  float rotax, rotx, roty;
  AplicaBoveda();
 public:
  BovedaCeleste(float, RtPoint, RtPoint, float, float, float, float, RtColor);
};

#endif


/*****************************************************************
 Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Modelo atmosferico.
Archivo bovedaceleste.C
Miguel Miranda Miranda.

Modificacion: Domingo 10 de junio de 2001.

Las tranformaciones en el shader cielo son para presentar una vista
logica del horizonte en las imagenes. Primero se deben de hacer las mismas
transformaciones en orden inverso a las que se le aplican a la camara
para que el espacio de shading de la boveda celeste quede centrada en la
posicion de la camara. La unica excepcion es la rotacion en el eje x que
concuerda con la vista del horizonte. Si un borde de la boveda celeste
aparece en la imagen no debe haber rotacion alguna en el shader pero
si no aparece se debe rotar la boveda para no obtener vista del horizonte.
Entre mayor sea la rotacion mas uniforme sera el azul del cielo.

*****************************************************************/

#include "bovedaceleste.h"

BovedaCeleste::BovedaCeleste(float tam, RtPoint pos, RtPoint dir, float rx, float
ry, float dec, float dens, RtColor col) {
  radio = tam;
  posicion[0] = pos[0];
```

```
  posicion[1] = pos[1];
  posicion[2] = pos[2];
  direccion[0] = dir[0];
  direccion[1] = dir[1];
  direccion[2] = dir[2];
  rotx = rx;
  roty = ry;
  decremento = dec;
  densidad = dens;
  color[0] = col[0];
  color[1] = col[1];
  color[2] = col[2];

  /* Inicializacion ajena al usuario final */
  iniciointegral = 0;
  finintegral = 20;
  k = 70;
  incrementomax = 0.5;
  incrementomin = 0.001;
  iluminacion = 1;
  debug = 0;
  casts_shadows = "none\0";

  AplicaBoveda();
}


BovedaCeleste::AplicaBoveda() {
  RiDeclare("densidad","float");
  RiDeclare("decremento","float");
  RiDeclare("iniciointegral","float");
  RiDeclare("finintegral","float");
  RiDeclare("incrementomin","float");
  RiDeclare("incrementomax","float");
  RiDeclare("k","float");
  RiDeclare("iluminacion","float");
  RiDeclare("debug","float");

  /* Se coloca la semiesfera donde se aplica el modelo atmosferico*/
  RiAttributeBegin();

    /* Las siguientes transformaciones compensan el movimiento
       de la camara sobre la boveda celeste. Esto es necesario
       para que se ejecute correctamente el shader */

    RiTransformBegin();
      RiTranslate(posicion[0], -posicion[2], posicion[1]);
      RiRotate(-roty, 0.0, 0.0, 1.0);

      /* Si la camara se desplaza sobre el eje de las z debe
       de considerarse la rotacion sobre la boveda celeste */

      if ( (posicion[1] > 0) && (direccion[1] >= 0) ) {
      proyxz = sqrt(posicion[1]*posicion[1] + 300*300);
      rotax = 180.0 * asin( posicion[1]/proyxz) / PI;
      }
      else {
```

```
    rotax = rotx+5;
    RiRotate(-rotx, 1.0, 0.0, 0.0);
    }
    RiRotate(rotax, 1.0, 0.0, 0.0);

    RiAtmosphere("cielo",
"iniciointegral",&iniciointegral,"finintegral",&finintegral,
"k",&k,"incrementomax",&incrementomax,"incrementomin",&incrementomin,"decremento
,&decremento,"densidad",&densidad,"iluminacion",&iluminacion, RI_NULL);
   RiTransformEnd();

   RiColor(color);
   RiAttribute("render", "casts_shadows", &casts_shadows, RI_NULL);
   RiSphere(radio,radio,0,360,RI_NULL);
 RiAttributeEnd();
```

El siguiente es el programa que integra todos los elementos. Se invoca a la geometría del ambiente virtual. El cielo se agrega creando una instancia del objeto BovedaCeleste La principal fuente de luz es el "shader" que simula la luz solar. El modelo de cámara que permite tomar diferentes vistas del ambiente se incluye como una instancia de la clase Camara.

```
/*****************************************************************
 Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Programa de aplicacion para el modelo atmosferico. Declara la
Funcion que incorpora la geometria del ambiente.
Archivo irixparte1.h
Miguel Miranda Miranda.
*****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ri.h>
#include "camara.h"
#include "bovedaceleste.h"

#ifndef __IRIXPARTE1_H
#define __IRIXPARTE1_H

extern void ciudad();

#endif


/*****************************************************************
 Universidad Nacional Autonoma de Mexico.
Instituto de Investigaciones en Matematicas Aplicadas y Sistemas.
Programa de aplicacion para el modelo atmosferico.
Archivo irixparte1.C
Miguel Miranda Miranda.
```

```
Modificacion: Sabado 9 de junio del 2001.
************************************************************/

#include "irixparte1.h"

/*****************************************************************
 Programa principal.
*****************************************************************/

int main(int argC, char** argV) {
  char   nombrearchivo[64];
  RtToken renderer = RI_NULL;
  RtColor color;
  RtPoint posicionsol;
  RtString encendido = "on\0";
  RtString casts_shadows = "none\0";
  RtString indirecto = "indirect.dat";
  RtFloat maxerror;
  RtFloat nsamples;
  RtFloat intensidad;
  RtLightHandle dia;
  RtLightHandle ambiente;
  RtPoint posicion, direccion;

  RtInt cuadro = 0;

  if (argC != 2) {
     #if defined(_WIN32)
         fprintf(stderr,
                 "Escriba: %s ArchivoRIB|rgl|rendrib\n\n",
                 argV[0]);
     #else
      fprintf(stderr,
              "Escriba: %s ArchivoRIB|rgl|rendrib|rendribv\n\n",
               argV[0]);
     #endif
     exit (-1);
  }

  renderer = argV[1];
  RiBegin(renderer);

    for (cuadro = 0; cuadro < 20; cuadro++) {
       sprintf(nombrearchivo, "AztecaInt%d.tif", cuadro );
       RiFrameBegin(cuadro);
          RiDisplay(nombrearchivo, RI_FRAMEBUFFER, RI_RGBA, RI_NULL);
       RiCropWindow(0.0, 1.0, 0.0, 0.5);
       RiFormat(600,400,1);
       RiProjection("perspective",RI_NULL);
       RiPixelSamples(1,1);
       //    RiOption("indirect", "savefile", &indirecto, RI_NULL);
       //    RiOption("indirect", "seedfile", &indirecto, RI_NULL);

         /* Se define la posicion y orientacion de la camara */
       posicion[0] = 100;
       posicion[1] = 60;
       posicion[2] = 100;
```

```
        direccion[0] = 1;
        //          direccion[1] = 0.5-(cuadro*0.05);
                direccion[1] = -0.5;
        direccion[2] = 1;
        Camara camara(posicion, direccion);

        RiWorldBegin();
           RiShadingRate(4);
           RiDeclare("intensity","float");
           RiDeclare("lightcolor","color");
           RiDeclare("sol", "point");
           RiDeclare("from","point");
           RiDeclare("to","point");
           RiDeclare("stepsize","float");
           RiDeclare("lightcolor","color");
           RiDeclare("sombras", "string");

           /* Se activan las sombras */
           RiAttribute("light", "shadows", &encendido, RI_NULL);

           /* Se indica la intensidad, color y posicion del sol */
           intensidad = 1.0;
           color[0] = 1.0;
           color[1] = 1.0;
           color[2] = 1.0;
           posicionsol[0] = 0;
           posicionsol[1] = 0;
           posicionsol[2] = -1;
           dia = RiLightSource("luzdia", "intensity", &intensidad, "lightcolor",
&color, "sol", &posicionsol, RI_NULL);

           /* Se aplica el modelo atmosferico */
           color[0] = 0.2;
           color[1] = 0.2;
           color[2] = 0.2;
           BovedaCeleste cielo(2000, camara.DamePosicion(),
camara.DameDireccion(), camara.DameRotacionX(), camara.DameRotacionY(), 200, 50,
color);

           /* Se aplica la interreflexion */
           /*
           RiDeclare("indirect", "float");
           RiLightSource("indirect", RI_NULL);
           maxerror = 0.125;
           nsamples = 100;
           RiAttribute("indirect", "maxerror", &maxerror, RI_NULL);
           RiAttribute("indirect", "nsamples", &nsamples, RI_NULL);
           */

           // Agrega la geometria de la ciudad.
           ciudad();

        RiWorldEnd();
      RiFrameEnd();
   }
   RiEnd();
   return 1;
```

'or ultimo listamos el archivo de Makefile que integra todos los elementos para generar un rchivo ejecutable.

```
 Makefile para el proyecto de Graficacion.

.C = g++
'LAGS = -g
)BJS = irixparte1.o ciudad.o camara.o bovedaceleste.o
;RCS = parte1.C ciudad.C camara.C bovedaceleste.C
)INCLUDES =
)LIBS =
.IBS = -lribout
:XE  = parte1


$(EXE): $(OBJS)
      $(CC) $(FLAGS) -o $(EXE) $(OBJS) $(LIBS)
.C.o:
      $(CC) -c $<
clean:
      rm -f $(OBJS) $(EXE)
```

Bibliografía.

[1] Anthony A. Apodaca. *Using RenderMan in Animation Production.* SIGGRAPH 1994 Course 4, August 1995.

[2] Anthony A. Apodaca, Larry Gritz. *Advanced RenderMan. Creating CGI for Motion Pictures.* Morgan Kaufmann Publishers, 1999.

[3] Anthony A. Apodaca, M. W. Mantle. RenderMan: Pursuing the Future of Graphics. *IEEE Computer Graphics & Applications*, volume 10(4), pág. 44-49. IEEE, July 1990.

[4] Anthony A. Apodaca, Darwyn Peachey. *Writing RenderMan Shaders.* SIGGRAPH 1992 Course 21, July 1992.

[5] James F. Blinn. Simulation of Wrinkled Surfaces. *Proceedings of SIGGRAPH 1978*, pág. 286-292. ACM SIGGRAPH, ACM Press, 1978.

[6] Michael F. Cohen, Eric Chen Shenchang, John R. Wallace, Donald P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. *Computer Graphics*, volume 22(3), pág. 75-84, 1988.

[7] Robert L. Cook. Shade Trees. *Computer Graphics Proceedings of SIGGRAPH 1984*, volume 18(3), pág 223-231. ACM SIGGRAPH, ACM Press, July 1984.

[8] Robert L. Cook. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics*, volume 5(1), pág. 51-72. ACM TOG, ACM Press, January 1986.

[9] Robert L. Cook, Loren Carpenter, Edwin Catmull. The REYES Image Rendering Architecture. *Proceedings of SIGGRAPH 1987*, pág. 95-102. ACM SIGGRAPH, ACM Press, 1987.

[10] Robert L. Cook, Thomas Porter, Loren Carpenter. Distributed Ray Tracing. *Proceedings of SIGGRAPH 1984*, pág. 137-145. ACM SIGGRAPH, ACM Press, 1984.

[11] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley. *Texturing and Modeling.* AP Professional, second edition, 1998.

12] James D. Foley, Andries van Dam, Steven K, Feiner, John F. Hughes, Richard L. Phillips. *Computer Graphics: Principles and Practice.* Addison-Wesley, second edition, 1990.

13] James D. Foley, Andries van Dam, Steven K, Feiner, John F. Hughes, Richard L. Phillips. *Introducción a la Graficación por Computador.* Addison-Wesley Iberoamericana, 1996.

14] Larry Gritz. *Blue Moon Rendering Tools.* World Wide Web, http://www.bmrt.org, 2000.

15] Larry Gritz, James K. Hahn. BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools*, volume 1(3), pág. 29-47, 1996.

16] David Halliday, Robert Resnick, Kenneth S. Krane. *Física. Volumen 2. Cuarta edición.* CECSA, 1994.

17] Eugene Hecht, Alfred Zajac. *Optica.* Addison-Wesley Iberoamericana, 1986.

[18] Kenneth I. Joy, Charles W. Grant, Nelson L. Max, Lansing Hatfield editors. *Tutorial: Computer Graphics: Image Synthesis.* Computer Society Press, 1988.

[19] Kazufumi Kaneda, Takashi Okamoto, Eihachiro Nakamae, Tomoyuki Nishita. Photorealistic Image Synthesis for Outdoor Scenery under various Atmospheric Conditions. *Visual Computer*, volume 7(5), pág. 247-258, 1991.

[20] Tomoyuki Nishita, Yasuhiro Miyawaki, Eihachiro Nakamae. A Shading Model for Atmospheric Scattering considering Luminous Intensity Distribution of Light Sources. *Computer Graphics*, volume 21(4), pág 303-310, July 1987.

[21] Tomoyuki Nishita, Eihachiro Nakamae. Continous Tone Representation of Three-Dimensional Objects Illuminated by Sky Light. *Proceedings of SIGGRAPH 1986*, pág. 125-132. ACM SIGGRAPH, ACM Press, 1986.

[22] Ken Perlin. An Image Synthesizer. Proceedings of SIGGRAPH 1985, pág. 287-296. ACM SIGGRAPH, ACM Press, 1985.

[23] Ken Perlin. *Making Noise.* World Wide Web, http://www.noisemachine.com/talk1, 1999.

[24] Pixar. *The RenderMan Interface. Version 3.1.* Pixar. http://www.pixar.com/products/rendermandocs/toolkit/Toolkit/index.html, September 1989.

[25] Pixar. *PhotoRealistic RenderMan 3.8 Release Notes.* World Wide Web, http://www.pixar.com/products/rendermandocs/toolkit/Toolkit/notes3-8.html, 2000.

[26] Pixar. *The RenderMan Interface. Version 3.2.* Pixar. http://www.pixar.com/products/rendermandocs/toolkit/Toolkit/index.html, July 2000.

[27] Raymond A. Serway. *Física. Tomo II. Cuarta edición.* McGraw-Hill, 1997.

[28] Steve Upstill. *The RenderMan Companion.* Addison-Wesley Publishing Company, 1990.

[29] John R. Wallace, K. A. Elmquist, Eric A. Haines. A Ray Tracing Algorithm for Progressive Radiosity. *Computer Graphics*, volume 23(3), pág. 315-324, July 1989.

[30] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques.* ACM Press, 1992.

[31] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of ACM*, volume 23(6), pág 343-349 ACM, ACM Press, June 1980.